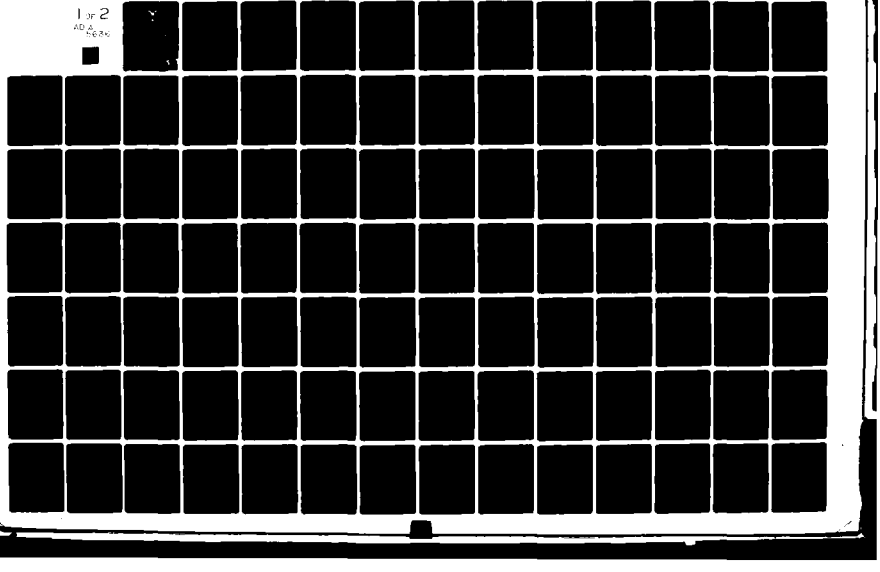
AD-A115 636 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH F/G 9/2
ANALYSIS AND DESIGN OF INTERACTIVE DEBUGGING FOR THE ADA PROGRA--ETC(U)
NOV 81 R L GAUDINO
UNCLASSIFIED AFIT/GCS/MA/81D-3 NL



UNITED STATES AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/MA/81D-3

ANALYSIS AND DESIGN OF INTERACTIVE DEBUGGING FOR THE ADA PROGRAMMING SUPPORT ENVIRONMENT

THESIS

AFIT/GCS/MA/81D-3 Richard L. Gaudino

ANALYSIS AND DESIGN OF INTERACTIVE DEBUGGING FOR THE ADA PROGRAMMING SUPPORT ENVIRONMENT

THESIS

PRESENTED TO THE FACULTY OF THE SCHOOL OF ENGINEERING
OF THE AIR FORCE INSTITUTE OF TECHNOLOGY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTERS OF SCIENCE IN COMPUTER ENGINEERING

BY

RICHARD L. GAUDINO

CAPT USAF

GRADUATE COMPUTER SCIENCE

21 November 1981

PREFACE

In an effort to significantly reduce costs for programming, the Department of Defense has funded the development of the Ada programming language and an Ada Programming Support Environment. This environment is designed to support the development and maintenance of software. A part of this development is to include a debugger facility for Ada programs.

The debugging facility specified in the Ada environment is designed to assist users in detecting, locating, and correcting errors in Ada programs. I felt that debugging facilities have been largely ignored in the past which gave me a strong interest and desire to do this thesis topic.

I would like to thank my advisor, Capt Roie Black, for all the help he gave me. His ideas and directions were essential. I would also like to thank my wife whose patience, help, and understanding made this thesis effort possible.

Table of Contents

List of Figures

ABSTRACT

This thesis involved the design and implementation of a skeletal interactive Ada debugger on the DEC-10 computer located at the Air Force Wright Avionics Laboratory. An analysis of current debugging technology was performed to formulat a basis for the debugger tool development. The tools implemented where breakpoints, single step and multi-step execution, display and modify program variables, as well as other miscellaneous options. Two conclusions were developed as the result of this thesis effort. First, because of the lack of information on current software debugging methods, I have concluded that more emphasis is needed in techniques and tools for debugging of programs. Second, I have concluded that more emphasis is needed in the human interfacing techniques.

1. INTRODUCTION

The Department of Defense has been forced to search for budget reducing strategies because of the ever increasing costs of software. An important goal of the DoD is the achievement of cost-effective reliability in the continuing support of long-lived embedded computer systems (Ref 2 : 6). An outgrowth of this search for cost-effective reliability was the development of the computer language Ada.

ADA is not just another "reliable enough" software language. During the development of the ADA language, the support environment for this language was also addressed. The ADA Programming Support Environment (APSE) objectives were developed to permit testing and debugging of ADA programs executing in any machine which is supported by an APSE (Ref 4 : 9-3).

A Mimimal Ada Programming Support Environment(MAPSE) tool set was also established because of a requirement for the portability of both the APSE tools and Ada. This MAPSE specified that a debugging facility to assist users in detecting, locating, and correcting errors in Ada programs was to be developed (Ref 2 : 3-33).

As one of many ongoing AFIT efforts with Ada, the subject of this thesis is the diagnosis of errors in ADA computer programs. This diagnosis is commonly referred to as software debugging. The purpose of this thesis is to gather

information on software debugging techniques and procedures available and then integrate these techniques as the basis for the development of a skeletal interactive debugging tool for the ADA programming support environment.

The interactive debugging tool for this thesis was developed on the Air Force Wright Avionics DEC-10 computer using the compiler developed by Capt Allen Garlington as one of the AFIT efforts with Ada (Ref 5).

1.1 OBJECTIVE

The objective of this thesis is to develop an understanding for the development of a software debugging tool and to implement a skeletal interactive ADA debugger. The study will be accomplished in steps which are documented in the following chapters.

The first step consists of acquiring and analyzing information related to current software debugging techniques. Several existing debugging facilities in use in Air Force installations were examined and common facilities were identified.

The second step documents software debugging requirements specified by Stoneman which provides guidance on the functional processes inherent in debugging, selection of tools and information requirements for software debugging activities.

Finally, all this information is used to develop a step-by-step debugging procedure. This debugging procedure is directly applicable to the ADA language currently being developed as one of the ongoing efforts within AFIT.

The last step of this thesis effort is to list conclusions and recommendations based on the previous steps.

1.2 SCOPE

Debugging is a complex set of interelated tasks which are initiated at completion of the translation of a software design into a machine-executable format. The task of debugging continues into the operation and maintenance phase of the software life cycle. The objective of this thesis is to concentrate on those debugging activities which are applicable to the ADA language. To clarify the scope of this thesis, the following definitions are given.

1.2.1 Debugging

Debugging is the art of locating an error once its existence has been established (Ref 11: 176).

1.2.2 A Debugging Tool

A debugging tool is a hardware or software (this thesis will deal only with software) capability used to find the symptoms and isolate the causes of an identified flaw. A debugging tool differs from a test tool in the purpose for which it is employed. A test tool finds software problems

while a debugging tool helps find the cause or causes of the problem (Ref 11: 177-178).

1.3 ASSUMPTIONS

The following assumptions limit the scope of this thesis:

1. No modification or enhancement to the Ada Compiler developed by Capt garlington will be made (Ref 5).

2. The computer environment to be examined for implementation is the DEC-10.

3. This thesis is concerned with error detection, not correction.

4. The approach taken, whenever possible, is general in nature even though this thesis is primarily intended for the ADA programming language.

5. No attempt is made to define any new debugging tool concepts. This thesis deals only with existing concepts.

6. Cost factors such as speed and memory will not be an issue of direct concern to any debugging capabilities.

7. This thesis implements only a partial ADA debugger since only a partial Ada compiler exists.

1.4 BACKGROUND

No one writes totally error free programs. In addtion to the problems of writing programs, programmers spend vast amounts of time debugging their source code. Estimates of the time spent on debugging vary from fifty to ninety percent (Ref 11: 119). A debugger would greatly reduce the time programmers spend correcting errors.

Computer programming is a creative art, best done by individuals, and debugging is a highly individual aspect of programming. The techniques that may be useful for debugging depend very highly on the individual situation: the environment in which the programmer is working, the debugging aids available, the particular algorithm being performed, and the nature of the particular bug or bugs the programmer is trying to find, Furthermore, in order to find a bug in a program requires some intelligence, i.e., a mechancial procedure can only occasionally tell what the error is. For example, the programmer may be informed that in the test program an attempt was made to access memory outside the range of an array. This may be the exact bug, but more than likely it is a symptom of a mistake elsewhere in the program which no mechancial procedure could determine.

Because debugging is such a highly personal and intellectual process, it has received very little detailed treatment in computer science literature in the past. It is

interesting to note that an examination of many introductory programming texts revealed that debugging is often not mentioned, while those texts that mention debugging only offer a few brief sentences. Even the book Program Style, Design, Efficiency, Debugging,and Testing by Van Tassel which offers a chapter on program debugging includes "added emphasis on getting the programs right the first time" (Ref 11).

There are several reasons that may account for debugging receiving only brief mentions in literature in the past. One reason is that of priorities. It has been suggested that debugging is the third of the major bottlenecks to programming and could not have received much attention until the first two were removed. The first bottleneck was hardware. During the early years of programming, a programmer spent much planning time squeezing programs into a limited amount of memory. This bottleneck has been greatly reduced in the past years with improved hardware with sufficient amounts of core. The next bottleneck was software for program writing. The developments in programming languages during the last decade have greatly eased the programmer's task of expressing algorithms in terms acceptable to the computer (Ref 6 : 28-31). We have, then, machines big enough and fast enough to execute most useful programs, and software varied and reliable enough to get those programs written; what now? Now; we get ADA which specifies an environment that includes

a debugger program requirement.

2. SOFTWARE DEBUGGING TECHNIQUES

The analysis of current debugging technology is a prerequisite to formulating a methodology applicable to development of an interactive Ada debugger. The choice of debuggers for analysis was based on the immediate accessibility of the debugging tools within the chosen facilities. Much of the information presented in this chapter is detailed in its examination of debugging tools and techniques. The reason for including such detailed information is to provide a firm basis for the subsequent tasks of this thesis. The tools selected to be analyzed are found on the following systems:

2.1 Multics

Multics H6180 located at the Rome Air Development Center (RADC), Griffiss AFB, Rome, New York.

2.2 GCOS

GCOS H6180 located at the Rome Air Development Center.

2.3 CDC

CDC 175/74 located at the Aeronautical System Division (ASD), Wright-Patterson AFB, Dayton, Ohio.

2.4 DEC-10

DEC-10 located at the Air Force Wright Avionics Laboratory (AFWAL), Wright-Patterson AFB, Dayton, Ohio.

2.5 Multics Debugging Capabilities

The Multics environment supports Fortran, APL, Basic, Cobol, and PL/1 languages. The debugging capabilities available for these languages on the H6180 Multics are Debug, Dump, Page Trace, Probe, Profile, Trace, and Trace Stack (Ref 8).

2.5.1 Debug

The Debug command is an interactive debugging aid that allows the user to look at, or modify, data and code. The concept of breakpoints is implemented and thus makes it possible for the user to gain control during program execution for whatever reason. Symbolic references permit the user of Debug to refer to variables of interest directly by name. The Debug command provides the user with the capabilities of looking at data or code, modifing data or code, setting breakpoints, performing transfers, calling procedures, tracing stacks, looking at subprogram arguments, controling and coordinating breakpoints, changing the stack, and printing the machine registers.

2.5.2 Dump

The Dump command prints selected portions of a segment in octal format. Dump will print out four or eight words per line and can be instructed to print out an edited version in ASCII representation.

2.5.3 Page Trace

The Page Trace command prints a recent history of page faults and other systems events.

2.5.4 Probe

The Probe command provides symbolic, interactive debugging facilities for programs compiled with PL/1, Fortran, or Cobol. Probe provides the following functions:

1. Set a breakpoint before or after a statement.
2. Call an external procedure.
3. Transfer to a statement.
4. Stop the program.
5. Assign a value to a variable.
6. Examine a specified statement.
7. Locate a string in the program.
8. Delete one or more breakpoints.
9. Display source statements.
10. Trace one statement and halt.
11. Advance one statement and halt.
12. Display the value of a variable.

2.5.5 Profile

The Profile command prints information about the execution of each statement in P1/1 or Fortran programs.

2.5.6 Trace

The Trace command monitors all calls to a specified set of externally compiled procedures.

2.5.7 Trace Stack

Many computers support the use of standard LIFO (last-in,first-out) stacks. These stacks can generally be

broken into data stacks and return stacks. The data stack is used to store numbers and addresses. The return stack is used to store program flow-control parameters. As the program executes, the information in the stack will change depending on the source code.

The Trace Stack command prints a detailed explanation of the current process's stack history in reverse order (most recent first). For each stack, all available information about the procedure which establishes the stack, the arguments to that procedure, and the condition handlers established in the stack are printed. Trace Stack also prints the machine registers at the time of a fault with an explanation of the fault and the source line in which the fault occured.

2.6 GCOS Debugging Capabilities

The debugging capabilities available on the H6180 GCOS are RBUG, TDS, Debug, Trace, and FDUMP (Ref 7).

2.6.1 RBUG

RBUG is a conversational debugging tool for batch programs using a teletype terminal as the programming device. The functions available with RBUG aid the programmer in inspecting and modifing program instructions, registers, and data parameters while testing the program in the execution environment. RBUG provides functions to print the registers, insert breakpoints, remove breakpoints, modify a register,

and terminate a program.

2.6.2 TDS

TDS is used to checkout and test a subsystem. TDS allows the user to gain control from a terminal at selected locations. TDS provides the user with the capabilities to insert breakpoints, delete breakpoints, modify a register, snap or display memory, display registers, and set and reset values.

2.6.3 DEBUG

Debug is an upgrade of RBUG and TDS. In addition to the functions of RBUG and TDS, Debug offers a trace mechanism, octal-to-decimal conversion, and decimal-to-octal conversion.

2.6.4 TRACE

The Trace package provides program information dynamically as it is executed. The trace shows the step-by-step operations of execution. The trace package allows for tracing the regsters, instruction execution, specified operation codes, and subroutines.

2.6.5 FDUMP

The FDUMP subsystem operates under a time-sharing system and is used to inspect and modify mass storage files in the permanent file system of GCOS. FDUMP will display a file in 320-word blocks and permit a word in the block to be modified.

2.7 CDC Debugging Capablities

Control Data Corporation offers the Cyber Interactive Debug (CID) Facility for Fortran Programming (Ref 3).

2.7.1 CID

CID is an interactive debugger that operates on Fortran programs. The functions available with CID aid Fortran programmers in inspecting and modifying program data parameters while testing the program in the execution environment. CID provides functions to set breakpoints, set traps, display variables, and alter variables.

2.8 DEC-10 Debugging Capabilities

TRW developed a Jovial Interactive Debugger (JID) which operates on the AFWAL DEC-10 computer (Ref 10).

2.8.1 JID

JID is an interactive debugger that operates on Jovial programs. The functions available with JID aid Jovial programmers in examination and modification of variables and execution of programs or procedures. JID provides functons to set breakpoints, display variables, alter varables, backtrace, and a help function.

2.9 Analysis

The debugging techniques listed in this chapter fall into three classes. These techniques can be classified as : 1)debugging with a storage dump, 2) debugging according to

traces, and 3) debugging with automated debugging tools.

Debugging by analyzing a storage dump is a very inefficient method. This technique uses a crude display of all storage locations in octal or heexadecmal format. Using the dump technique presents many problems. The correspondence between storage locations and the variables in the source program is difficult to establish using dumps. Dumps provide a massive amount of data, most of which is irrelevant. The dump technique only provides a static picture of the program and is rarely produced at the exact point of the error. Therefore, the program's state at the point of the error is not shown and actions taken by the program between the time of the error and the time of the dump can mask out required information. Finally, there is no set methodology for finding the cause of the error by analyzing a storage dump (Ref 11 : 213).

The second debugging technique, program traces, is not much better than dumps. Traces are superior to the use of dumps in that the trace displays the dynamics of a program and allows the programmer to examine information that is much easier to relate to the source program. However, traces have many problems. The trace can result in a massive amount of data to be analyzed. This does not encourage the programmer to think about the problem being debugged but rather provides a hit-or-miss method of finding errors (Ref 11 : 213-215).

The final debugging technique, automated debugging tools, allows the analysis of the program by using debugging features of special interactive debugging tools. A common function of the debugging tools is the ability to set breakpoints causing the program to be suspended when a particular statement is executed, or when a particular variable is altered. These features of the debugging tools allow the programmer to examine the current state of the program. This method is not an automatic feature for correcting errors but it does encourage the programmer to think about the problem being debugged (Ref 11 : 215-216).

3. REQUIREMENTS FOR THE ADA DEBUGGER

3.1 Tool Selection/Requirements

The Minimal Ada Programming Support Environment (MAPSE) design specified that the debugging facilities for ADA were to assist the users in detecting, locating, and correcting errors (Ref 2 : 3-33). The MAPSE design specifically stated that the Ada debugging facilities would support all Ada language features, including the following features as a minimum.

3.1.1 Breakpoints

The breakpoint feature for Ada debugging is to include conditional, preset, dynamically set, and single step breaks (Ref 2 : 3-33).

3.1.2 Display Values

The display feature for Ada debugging is to include displaying variables as well as constants (Ref 2 : 3-33).

3.1.3 Modify Values

Like display, the modify feature is to include modifing variables and constants (Ref 2 : 3-33).

3.1.4 Display and Modification

Display and modification of variables in machine representation (such as hexadecimal) or scalar type representation based on the user's option (Ref 2 : 3-33).

3.1.5 Display Subprogram Arguments

The values of each argument or formal parameter of subprograms (procedures) will be displayed (Ref 2 : 3-34).

3.1.6 Modify the Flow

The flow of the program will be able to be modified using a jump option (Ref 2 : 3-34).

3.1.7 Tracking

A tracking of the program by listing changes in variables as well as specifing the executing statememts shall be performed (Ref 2 : 3-34).

3.1.8 Dumps

A Dump option shall be included (Ref 2 : 3-34).

3.2 The Ada Compiler

Because debuggers are directly tied to a particular compiler, the first step toward developing an Ada Debugger is to establish an understanding of the Ada Compiler. The Ada compiler that is used for this thesis was developed by Alan Garlington as part of his thesis effort in the design and implementation of an Ada pseudo-machine (Ref 5). Several important features of the compiler need to be understood before the development of the debugger can be considered. These include the symbol table, pseudo-code, stack implementation, and nesting and recursion.

The technique used by the Garlington compiler was the

development of the pseudo-code compiler. This compiler generates pseudo-code for a hypothetical processor that enhances the compiler's portability. The pseudo-code generated from this compiler is then executed by an interpreter program which runs on the actual processor (Ref 5 : 8-9).

An interface between the compiler and the Debugger must be made since the debugger uses output from the compiler. The compiler generates the pseudo-code from the programmer's source code. The debugger must have a link between the source code and the compiler generated pseudo-code. This link is necessary for the proper execution of the debugger program.

3.2.1 Pseudo-Code

The Garlington Ada compiler was written using techniques that enhanced its portability. The technique that was used was the pseudo-code compiler approach. Using this approach, the compiler generates pseudo-code for a hypothetical processor. The instruction set was designed to be executed by an interpreter program which runs on an actual processor. Since there was no pseudo-machine, a simulator was created. The simulator program accepts the pseudo-code as input and then accomplishes the necessary actions for each instruction of the pseudo-code (Ref 5 : 8-9).

The instruction set output from the compiler was specifically developed to meet the needs of the Ada language

although only a subset of the language was considered. The current instructions implemented are divided into six classes: relational operators, single-word loads and stores, tasking operators, integer arithmetric operators, input/output operators, and miscellaneous operators. Each instruction contains three fields: the operation code, the level, and the address fields.

The operation code field contains the name of a specific operation to be performed. At simulation time, these operands are implemented on the run-time stack and evaluated according to the indicated operation.

3.2.2 Symbol Table

A copy of each identifier appearing in a program is contained in a string table. Linked to this string table is an attribute table which contains the properties assigned to each identifier. The attributes can then be made available throughout the compilation process. Together these two tables are called the symbol table.

In the generation of the pseudo-code, the compiler generates a symbol table which contains a description of all user-defined symbols relevant to the environment in which they are declared (Ref 5 : 61).

An analysis of the symbol table is extremly important to formulating a methodology for the development of the

debugger. The symbol table which is used by the compiler to generate the pseudo-code is also needed by the debugger so that it can manipulate program variables.

The compile time symbol table contains information which pertains to the type of data that appears within the source program. The table contains information on variables, arrays, subprogram names, and constants. The compiler does not generate a fixed run-time address for these symbols at compile time. The symbols are assigned locations relative to the beginning of the procedure (this can be done since the storage space required is known) (Ref 5 : 37).

3.3 Stack Implementation

In a block structured language such as Ada, each time a given procedure is activiated, the procedure is allocated a fixed amount of storage on top of a data stack. The size of the stack is implementation dependent. The amount of storage allocated will depend on the declarations in the procedure (or block), but the total storage needed will be computed at compile-time. This storage remains assigned to that procedure as long as it is activated. When an exit of a procedure occurs, that storage assigned to the procedure will be freed for reuse. Using this method, the total amount of storage allocated for data is limited by the number of procedures currently activated. The data allocation is therefore independent of the total procedures or blocks used

in the program. Using this scheme, the particular section of the stack in which the data area of a procedure may reside will vary during program execution and cannot be predicted at compile-time. Each activation of the same procedure may be in an entirely different section of the stack.

The storage area will hold the values of the variables and task activation information. The task activation information includes the stack-frame control data. The values of all the variables are determined at run-time (see Figure 3-1). Constants are fixed at compile time and cannot be changed for successive activations of a procedure.

The task activation information is generated upon entrance to a block, subprogram, or upon initialization of a nested task object. The stack-frame contains the static link, the dynamic link, program counter, task flag, exceptions, priority, top of stack, base, link, heap, data lock, caller, return, and entry (Ref 5 : 30-1). Each of these terms are defined in the Appendix.

The physical location of an identifier will change during execution, but the relative offset from the beginning of the area reserved for the procedure in which it is declared will be fixed. If the location of the beginning of the procedure's data area is known, the location of the identifier's storage can then be computed at run-time. Figure 3-3 shows the organization of the data area for the program

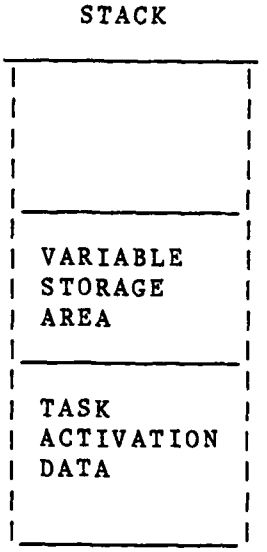


Figure 3-1: STORAGE AREA

```
1       PROCEDURE MAIN IS
2       VAR
3         A, B : INTEGER;
4       BEGIN
5        A := 1;
6        B := 2;
7       END MAIN;
```

Figure 3-2: PROCEDURE MAIN

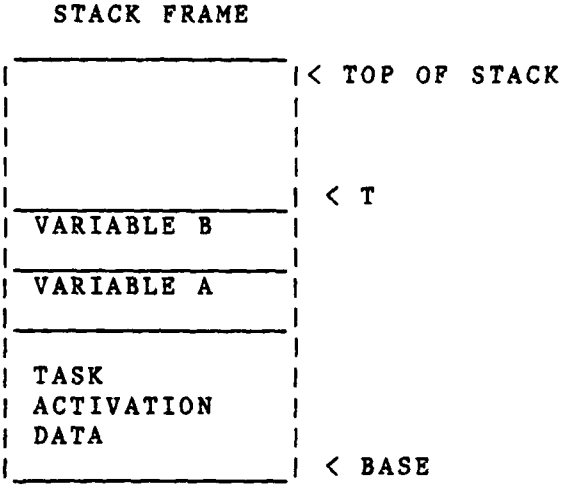


Figure 3-3: DATA AREA ORGANIZATION

	OPERATION	LEVEL	OFFSET
1	INCT	0	18
2	ILOADCONST	0	1
3	ISTORE	0	17
4	ILOADCONST	0	2
5	ISTORE	0	18

Figure 3-4: PROGRAM MAIN COMPILED CODE

Main of Figure 3-2. The size of the task activation area is 16 (Ref 5). The storage for variable A is offset 17 words from the beginning of the data area. Variable B begins one word after the location of A. The compiler output for program Main would be as seen in Figure 3-4. At run-time, storage space for the task activation and variable areas are reserved by the Increment command allowing an area of 18 words. Integer one would be loaded on top of the stack by the ILOADCONST. The ISTORE command would then cause the data on top of the stack (one) to be stored in location 17 or the

space reserved for A. Likewise, B would have the value two stored in the location 18.

3.3.1 Nesting and Recursion

A separate storage area is created every time a new procedure or block is activated (i.e. a call). The new storage area, which consists of the task activation and variables, can be referenced from the new active procedure. In defining the nesting levels of a procedure, the nesting level would be one plus the nesting level of the procedure that performed the activation. If the nesting level of the old data area was one then the new nesting level would of course be two. This nesting level will be refered to in terms of the activation frame.

With a storage area included for every procedure, referencing the new data area is performed by adding the offset (identifier's address) to the new base. The base is set to the beginning of the storage area location for the procedure (see Figure 3-5). Each time a procedure is activated, a new activation frame is added. Program variables defined in the main program are commonly referred to as global variables. The global variables can be referenced from any procedure whereas procedure variables can only be referenced within the given stack activation frame and are "invisible" to all other procedures.

The task activation area is used to link together each

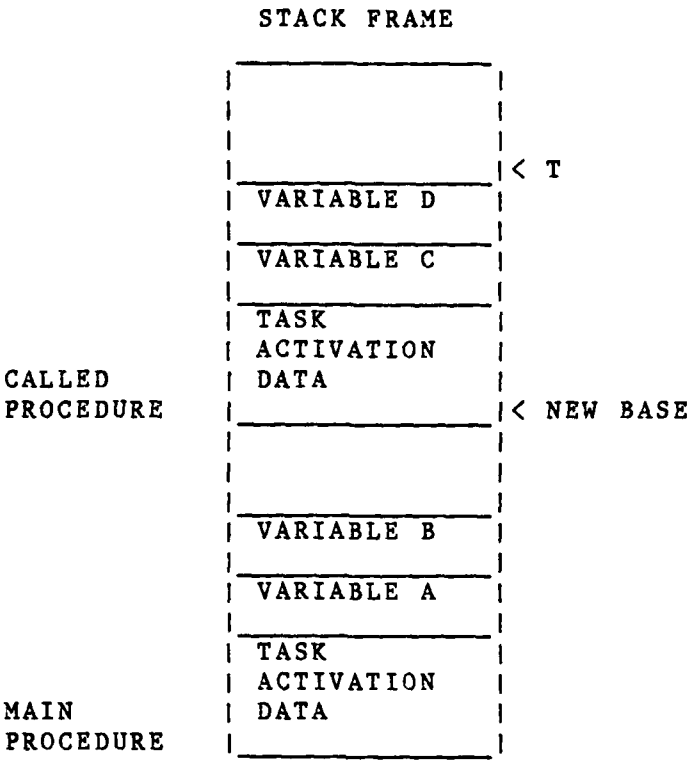


Figure 3-5: STACK WITH A CALLED PROCEDURE

activation frame. Whenever execution enters a new procedure of the program, another activation record is allocated in the stack frame and "pushed" on to the stack. In order to assure returns to the calling procedure, link words are maintained in each activation frame. One link shows the static linking of the procedure. This link defines the nesting environments. The second thread which is the dynamic chain provides information necessary for the environment adjustments which include the deallocation of the activation frame when the executing procedure does a return. The information in the task activation area is defined as the STACK-FRAME Control

AFIT/GCS/MA/81D-3

Data in the appendix.

4. DEBUGGER TOOL IMPLEMENTATION

This chapter describes the tools implemented for the ADA debugger. It defines the interface between the Ada compiler and debugger along with an indepth discussion of each tool.

4.1 The Compiler Output

The Garlington Ada compiler was modified to add a debug switch which enabled the output needed from the compiler by the debugger to be written to a file COMCOD.TXT. This file has the pseudo-code, source-line table, and the symbol table. This output is only obtained when the debug switch is set during a compile.

4.1.1 Pseudo-code

The pseudo-code as described in Chapter 3 is of the form : OP LEVEL ADDR. This pseudo-code is used to perform the simulated execution using procedure INTERPT.

4.1.2 Source-line Table

In order to implement the different breakpoint options, a reference is needed between the source code and the pseudo-code. This reference is made in the compiler and output to the COMCOD.TXT file in the form of the source-line table.

The source line table consists of entries which have an index in the code table. Each entry in the source-line table refers to the last entry of the code table which was

generated by the previous source statement. For example, the procedure Main (Figure 3-2) consists of seven lines of source code. These seven lines of source code generate five lines of pseudo-code (Figure 3-4). The source-line table that would be generated for this program can be seen in Figure 4-1. These figures show that from this source code the compiler generates two instruction lines of pseudo-code for source line five (A :=1), that is the pseudo-code entries two (ILOADCONST) and three (ISTORE). The fifth entry into the source-line table enables the debugger program to note that after the execution of the pseudo-code indexed one, the pseudo-code for the source line five will begin.

SOURCE LINE	CODE INDEX	SOURCE LINE
1	0	PROCEDURE MAIN IS
2	0	VAR
3	0	A, B : INTEGER
4	0	BEGIN
5	1	A := 1;
6	3	B := 2;
7	5	END MAIN;

Figure 4-1: SOURCE-LINE TABLE

4.1.3 Symbol Table

The compiler data structure that associates identifiers with their attributes is called the symbol table. In order for the debugger to provide the tools of modify and display, the symbol table must be maintained to provide these

attributes.

The compiler uses the symbol table each time an identifier is referenced. The compiler generates new entries for the symbol table when a new identifier is declared. In a block-structured language like ADA, the scope of an identifier is that of the block in which it is declared. However, the same identifier can be declared several times within the same block without conflict.

This method of scoping is of great value to programmers. A new block of source may simply be inserted into an existing program. If a new temporary variable is needed in the new block, the variable may be declared within the block without any concern that its name may have already been used in another block.

The compiler handles this scoping by adding the identifier to the symbol table upon entry of a new block. When an exit of the block occurs, the identifier is removed from the symbol table and may not be refered to again.

The debugger needs the identifier to provide the display and modify tools. As each identifier is removed from the symbol table it is stored in a table during the compile. This table provides the attributes of each identifier. The attributes included in this table are the string identifiers, the type of the identifier (i.e.integer,constant), the

address or value, and the scope that the identifier was defined in. By saving the identifier in this table, the identifier is not thrown away by the compiler. On completion of the compile, this new symbol table is written to the file COMCOD.TXT for use by the debugger if the debug switch is set.

4.2 The Debugger Tools

Only a subset of the tools specified by the MAPSE were implemented as part of this thesis effort. The tools that were omitted are conditional breaks, display and modify constants, timing statistics, and the full implementation of the jump tool. These tools were ommitted due to time constraints. The tools that were included in the development of the debugger are as follows:

4.2.1 Breakpoints

The breakpoint tool was implemented to stop the program execution prior to the source line that the user requested. Once the breakpoint condition has been met, the user would have control of the program at the requested source line and could request any of the other tools offered by the debugger program or to continue until the breakpoint condition was met again.

To enable the user to set breakpoints, the program sends a message to the users terminal prior to execution of the pseudo-code. The user would then be able to specify the

source line at which simulation by the degugger should stop and return control. Checks were implemented in the breakpoint code to make sure that the breakpoint specified by the user was valid. For example, the user might mistype the source line number or may specify a line number that was not included in the source line table.

Once a breakpoint is set, the debugger program would continue execution of the pseudo-code. A check is made each time a new line of code is retrieved. The first check is performed by finding the source line that generated the current line of pseudo-code from the source-line table. The index value from the source-line table is then compared to the set breakpoint. If the index value matches the breakpoint, control is returned to the user, otherwise the new line of code is executed.

4.2.2 Single Step

The single step tool was implemented to stop the program execution prior to each new source line encountered. Since the single step stops prior to every new source line, the set breakpoint condition would be of no value. The single step tool therefore turns off all breakpoints if previously defined.

Once the single step condition has been met, that is pseudo-code for the next source line is encountered, the user would have control of the program. The user could then

request any of the other tools offered by the debugger program or continue in the single step mode.

To enable the user to select the single step mode, the program sends a message to the user's terminal prior to execution of the pseudo-code. The user would then be able to specify the single step option. The user would also be able to specify the single step option when a breakpoint was encountered. Selecting the single step option at this point, however, would turn off the breakpoint.

Once the single step option is set, the debugger will continue execution of the pseudo-code. A check is made each time a new line of pseudo-code is retrieved. The index to this code is compared to the breakpoint value. The breakpoint value is set to the index value in the source-line table each time a new source line is encountered. If a breakpoint matches the current pseudo-code index then control would be passed to the user, otherwise the new line of pseudo-code is executed.

The single step tool stops execution prior to each new source line. When a procedure or function is encountered, the single step tool will stop on the first line of source code of that procedure or function. When a return is encountered for the procedure, the single step tool will stop on the source line following the call.

4.2.3 Multi-step Execution

The multi-step tool was implemented to stop the program execution after a user specified number of source lines of code had been executed. Like the single step tool, the multi-step tool turns off all breakpoints if any were previously defined.

The user may specify the multi-step option prior to the execution of the program, after a breakpoint condition, or after a single step condition has been met. When the multi-step option is selected, the user must specify the number of steps to be executed before the program continues to execute.

Once the number of steps have been specified, the debugger program would continue execution of the pseudo-code. A check is made each time a new line of pseudo-code is encountered to determine if it is the beginning of a new source line. This check is done by comparing the index to the pseudo-code table to the source-line table. If there is a match, a line-number counter is incremented. If this counter matches the user's specified number of steps, control passes back to the user, otherwise execution continues.

4.2.4 Trace

The trace tool was implemented to show the user all changes that are made to program variables during execution. The trace tool can be selected prior to execution or at any

break which occured as the result of a breakpoint, single step, or multi-step condition being met. When the user selects the trace option, the trace flag is set to true.

When a store command is performed (i.e.ISTORE), the data item on top of the stack is placed into the location reserved for the variable by the method described in the section Stack Implementation of Chapter 3. If the trace flag is true, the new value for the variable is also displayed to the user.

Displaying just values to the user would leave it to the user to determine which identifier each value of the trace refered to. To avoid this annoyance, the debugger program displays the string representation for the identifier along with the new value. In order to display the correct string representation, the debugger program must determine the frame level, base, and address (offset) for the identifier being modified. After these items have been determined, the string representation is found in the symbol table based on the offset and level being altered. The string value found in the symbol table is then displayed to the user along with the identifier's new value.

4.2.5 Jump

The Jump tool was implemented to allow the flow of the program to be modified. The debugger jump option modifies the program counter to the user specified value.

The user may specify the jump option after a breakpoint, single-step, or multi-step condition has been met. When the jump option has been selected, the user must specify the source line number that the execution will continue on.

Once the source line number has been specified, the debugger program performs a check to determine if the line number given is within the range of possible values for the source program being executed. If the specified number is not in range, the debugger will default to the last line of code and set the termination flag.

The jump option only supports a jump within the current level of execution. The debugger program provides no checks for jumps outside the current level of execution. Checks were not implemented because of a lack of time to finish this necessary function. These checks are necessary since a jump outside the current level of execution would cause two possible errors.

The first error that could occur would be a jump into a procedure which has not been called yet or does not exist on the stack. Since the stack frame for this procedure is not initialized, all addressing of variables would be wrong.

The second type of error that could occur would be a jump into a procedure that is initialized within the stack but at a different level. Again, this error would cause

addressing errors as well as errors of deactivation of stack frame levels.

These errors could be removed by incrementing or decrementating the stack based on the location of the jump. Incrementing or decrementing could be done easily by issuing a dummy operation Call or Return for the procedure. Problems would occur using this method, however. In a block structured language such as ADA, the dynamic and static linking would have to be modified. This modification would vary based on the level for which the jump is intended for. Becuase of these problems, the jump option was not further implemented.

4.2.6 Display Values

The Display value tool was implemented to display specified program identifiers and their values to the user. The display tool can be selected at any break which occured as the result of a breakpoint, single step, or multi-step condition being met.

Once the debugger has received the display request, the user must specify the identifier to be displayed. The debugger program searches the symbol table as a check to make sure that the identifier exists. The debugger program also checks to determine if the specified identifier is defined in more than one level. If the identifier does not exists in at least one level, the debugger will display a message to the user that the identifier requested does not exist.

Once the debugger has determined that the variable exists in at least one level, the stack area is searched and compared at each frame level for the specified identifier. When the identifier has been found, its value along with its string representation and frame number is displayed. The debugger also checks for identifiers that are defined as constants.

If the identifier's block has not been activiated, the debugger will display the procedure name and the identifier with the message that the identifier has not been defined. If the identifier is defined in more than one environment or resides in a recursive procedure, all current values will be displayed.

4.2.7 Modify Values

The Modify value tool was implemented to allow the modification of program identifier values. The modify tool can be selected in the same manner as the display tool.

Once the debugger has received the modify request, the user must specify the identifier to be modified. The debugger program searches the symbol table as a check to make sure that the identifier exists. The debugger program also checks to determine if the specified identifier is defined in more than one level. If the variable does not exist in at least one level, the debugger will display a message to the user stating that the identifier requested is not currently

defined. This could occur if the procedure where the identifier is defined in not been activated yet or has been deactivated. If the identifier exists in more than one environment, the debugger displays a message to the user asking for the specific procedure for the modify.

Once the environment has been obtained, two more checks are performed. The first check is to determine the type of the identifier. Only integers and constants where implemented since the compiler only supports these types at this time. If the identifier is a constant a message is displayed to the user stating that modifying constants is not allowed. The compiler implemented constant values within the pseudo-code and could not be modified without changing the values on the pseudo-code file.

The second check was performed to determine if the requested identifier within a given environment was defined at more than one frame level (recursion). If the identifier was defined at more than one frame, the user is asked to specify which frame is to be modified. A check is performed to make sure the specified frame matches the frame the identifier is defined in. If the user specified a frame that the identifier is not declared in, a message is displayed to the user.

After all checks have been performed, the user is requested to specify the new value. This new value is then

inserted into the proper frame and address of the stack.

4.2.8 Modify Program

Since it is not the function of the debugger to modify the program, the modify program tool was implemented to show the feasibility of linking to the Editor program.

The Modify program tool allows the Editor to be linked so that the program could be modified. The Ada Editor has not been developed so a dummy Editor program was linked to prove that the linking would be performed.

The user may specify the modify program option after any break which occurs as the result of a breakpoint, single step, or multi-step condition being met. When the debugger program receives the modify program option a link will be made to the dummy Editor which will only terminate execution after displaying a message that the link was made.

4.2.9 Output Stack

A dump tool was implemented to display all information that resides in the stack area. This output option displays the current process's stack history in the order of the first implemented to the most recent frame implemented.

The user may specify the output stack option after a breakpoint condition, single step condition, or multi-step condition has been met. When the output stack option is selected, the stack as well as the address location is

displayed to the user.

4.2.10 Miscellaneous Tools

The debugger program provides two additional tools. A help option and block entry/exit display are provided to assist the user.

The help option specifies the options available to the user. This option prints out the commands and their functions.

The block entry/exit displays the block name to the user each time a new block is entered and another display when an exit of a block occurs. These features help the user keep track of which block the program is currently executing.

5. RECOMMENDATIONS AND CONCLUSIONS

This Chapter describes the deficiences of the debugger and also describes some areas where continuation efforts could begin. The recommendations are divided into three parts. The first section in the chapter describes improvements to the debugger as the compiler changes. The second section describes improvements which could be made to the debugger tools. The third section describes suggested improvements to the man-machine communications techniques. The final section of this chapter provides conclusions of this thesis effort.

5.1 Improvements as the Compiler Changes

The compiler used by the debugger needs to include additional tool development. Implementation of types and subtypes other than just integers need to be included into the compiler and debugger. Separate compilation as well as overloading should also be implemented.

As an Ada Programming Support Environment is developed for AFIT, considerations must be made to permit all tools of the Ada language to work together. Because of these ongoing AFIT efforts, one central data base area should be established and defined to integrate the compiler, debugger, editor, as well as the linker loader.

5.2 Improvements to the Debugger

The current debugger allows only one breakpoint to be set at a time. Implementing multiple breakpoints would be a beneficial improvement. Multiple breakpoints could be implemented easily using an array for each breakpoint.

Multiple options at any given time should be included as an improvement. Currently only one option at any given time can be requested. The multi-step and breakpoint tools would be more useful if these tools could be implemented concurrently.

The Jump option should be improved to allow jumps outside the current activation frame. This improvement would allow greater flexibility in modifing the flow of execution. To allow this improvement, task activation and deactivation will be needed as part of the jump option.

Finally, the Modify program option will need to include the incorporation of the Editor program. When the Ada Editor has been developed a link should be made from the Debugger to the Editor for any program modifications.

5.3 Improvements to the Man-machine Communications Interface

A bottleneck remains which hinders the effectiveness of the debugger. This bottleneck is the man-machine communications. The debugger is not very good at communicating with the user, which is a situation all to

common for the users of interactive systems.

Debuggers usually tend to be ineffective to the needs of the user. There are two important causes of the ineffectiveness of the debugger which is typical of interactive computer systems.

The first cause is that the debugger commands are a highly restricted artificial language designed specifically for use by the debugger. If the user fails to use these options or makes a mistake, however small, an error message plus a request to rephrase would be the best response that could be added. This inability of the debugger as well as current interactive systems to make such corrections is very frustrating for the human user.

A second cause of the ineffectiveness of the debugger is its poor performance in keeping track of the current context. People can suspend one context temporarily, switching conversation to a different context, and then going back to the orginal context, but the debugger can not.

Currently, the debugger as well as most other interactive programs have the man-machine interaction take place through typed input and output. The typical mode of operation is for the user and the system to take turns typing on a scroll of paper or scrolled display screen. However, this type of man-machine communication is rapidly becoming

outmoded by newer generations of powerful computers using LISP, FORTH, and SMALLTALK. These machines are intended for dedicated use by a single individual and feature a high-resolution, graphic display, as well as a conventional keyboard. This allows the computer to provide multiple independent output channels by dividing the screen into windows, in addition to the two independent input channels of keyboard and pointer (Ref 9 : 90-147).

In addition to the usual character output, a graphics screen can display line drawings or images and produce attention effects such as highlighting the background of certain areas of the screen. On the input side, a display object can often be referred to by pointing much more efficiently than by a typed description (Ref 1 : 20).

Using a graphic type screen, however; can cause difficulties in the management of the screen resources. The debugger output could divide the screen into windows which could be rearranged on the display like sheets of paper. This type of organization would be attractive since it allows a large amount of related information to fit onto the screen at one time. Parts of the windows could be selectively obscured and reused for other displays. In the management of these windows the user should always be able to arrange windows to see the parts of the displays that interest him. The debugger should also provide automatic position displays so the users

would not need to intervene. Finally, the screen should be exploited to provide the largest possible working space for the user while at the same time be kept simple enough to avoid confusion (Ref 1 : 28).

In addition to the graphic displays, other recent computer technology advances have been made. One of the more recent advances is in the area of pointing devices such as the joystick, mouse, or cat. These pointing devices allow the user to control the position of a cursor or the display screen to select specific objects that were displayed earlier. This facility would enable the user to add quick debugging commands (Ref 9 :9 8).

Finally, a method for flexible parsing would greatly enhance the man-machine interaction. Most current parsing systems do not allow for the possibility that the input might deviate slightly from the internal grammer and yet still be useful. An interface which discards all ungrammatical input appears very cumbersome to the user. Implementing a flexible parser that would allow for reasonable deviations could be an improvement. By allowing one or more ungrammatical inputs into the grammatical input would improve the communications between the user and machine (Ref 1 : 22).

5.4 Conclusions

Two significant conclusions have been developed as the result of this thesis effort. The first conclusion relates to

current software debugging information. The second conclusion relates to the DoD specified support environment.

In acquiring information related to current debugging techniques, it was very interesting to note that debugging receives only brief mentions in much of the literature. When mentioned, debugging is often mixed with testing which is a totally separate issue. This lack of information leads to the conclusion that even though emphasis is needed on getting the program right the first time, more emphasis is needed in techniques and tools for the debugging of programs.

In an effort to reduce programming costs, the Department of Defense has recognized the need for a support environment for the Ada language. This support environment makes Ada more than another "reliable enough" language. In addressing the support environment as part of Ada, DoD is establishing a new standard for total software develpoment. The choices of debugging tools which are to be included in the DoD's debugger include all the tool capabilities that are available in current interactive debugging technology. However, this leads to the second conclusion. DoD has not gone far enough in the specifications for the Ada environment. In the specifications for an Ada environment, the human interface to the computer should be emphasized not only for the debugger but for all of the Ada support tools.

BIBLIOGRAPHY

1. Ball,Eugene. Breaking the Man-Machine Communications Barrier. Computers 14 (1981), 19-29.

2. Computer Science Corporation. ADA Intregrated Environment System Specification. 1981.

3. Control Data Corporation. Fortran Version 5 Reference Manual. Sunnyvale, California, 1979

4. Defense Advanced Research Projects Agency. ADA Integrated Environment. Department of Defense, Washington, D.C., 1981.

5. Garlington, Alan R. Preliminary Design and Implementation of an ADA Pseudo-Machine. Master Th., Air Force Institute of Technology, March 1981.

6. Halpern, M. Computer Programming: The Debugging Epoch Opens. Computers and Automation 14 (1965), 28-31.

7. Honeywell Information Systems. Debug and Trace Routines; Series 600/6000 GCOS. 1972

8. Honeywell Information Systems. Programmer's Manual Volume III - Commands and Active Functions, Multics Software. 1973

9. Tesler, Larry. The Smalltalk Environment. Byte 16 (1981), 90-147.

10. TRW Defense and Space Systems Group. Computer Program Product Specification For Jovial Interactive Debugger. Redondo Beach, California, 1980

11. VanTassel, Dennie. Program Style, Design, Efficiency, Debugging, and Testing (Second Edition). Prentice-Hall, INC., New Jersey, 1979.

I. APPENDICES

I.1 Peculiarities of DEC-10 Pascal

Since there exists only a partial ADA compiler, Pascal was used to develop the debugger program. The debugger program was written using DEC-10 Pascal. The DEC-10 has features that differ from other versions of Pascal.

In DEC-10 Pascal, the terminal is a separate input channel from the standard file INPUT but is treated just as other input character files are, including a look-ahead file window feature. However, there are some peculiarities and drawbacks to using Pascal.

It is not possible to read any character, string, or number from the terminal without terminating a group of such numbers with a return, since this method is the only way that data is passed from the DEC-10 terminal line buffer to a Pascal program. Because the Return character will be the next following character after the data is read, a READLN(input) will not be satisfied with that terminating Return since the look-ahead character after the Return will not be read yet. Thus the READLN statement can never be used in conversational programming from the TTY, since Pascal will wait for still another input character beyond the Return. In addition, this additional input character is not passed to Pascal until another Return is typed since only lines are passed to Pascal, not characters. Thus the statement, READLN(TTY,A,B),

will wait for a second Return after reading A and B from the line passed by the first Return. To avoid this problem in interactive programming with the DEC-10 Pascal, the READ statement is used in all cases, rather than the READLN statement when reading from TTY.

I.2 DEC-10 Character Input

The DEC-10 Pascal system automatically puts a character in the look-ahead window of the file TTY when any program starts that will utilize reads from the terminal. At the time DEC-10 Pascal program is executed from the monitor, the program is linked and loaded and types an asterisk to indicate that it is ready for that first character of input. This is usually just a Return (which, of course, becomes a space when passed to Pascal).

This space in the file window is usually irrelevant if the first data to be entered from the terminal is to be a number of type Integer or Real, since leading spaces are always ignored. However, if the first data in the program is to be a character input from the terminal, it will be that space in the look-ahead buffer that is used unless this space is read and discarded. Consider the program of Figure 5-1.

Assuming that the user runs the program of Figure 5-1 by typing "EX EXAMPLE.PAS", when the program is loaded, the output to the TTY unit would be "*". The user would then type a return to start the execution followed by "ABC". The

```
PROGRAM EXAMPLE;
VAR A, B, C : CHAR;
BEGIN
 READ(TTY,A,B,C);
 WRITELN(TTY,'A =',A);
 WRITELN(TTY,'B =',B);
 WRITELN(TTY,'C =',C);
END.
```

Figure 5-1: INCORRECT PROGRAM EXAMPLE

```
PROGRAM EXAMPLE2;
VAR A, B, C, DUMMY : CHAR;
BEGIN
 READ(TTY,DUMMY);
 READ(TTY,A, B, C);
 WRITELN(TTY,'A =',A);
 WRITELN(TTY,'B =',B);
 WRITELN(TTY,'C =',C);
END.
```

Figure 5-2: CORRECT PROGRAM EXAMPLE

program output to the TTY unit would be:

```
A =
B = A
C = B
```

because the TTY window was loaded with the first character, a space, which was then read into variable A. Then variables B and C were loaded with the characters A and B, respectively. In any program where the first data to be read will be a character, a read must be made to throw away that extra

window character which is entered first.

The program of Figure 5-2 illustrates the correct use of the look-ahead character.

I.3 Linking Externally Compiled Programs

Linking of externally compiled programs can be performed using the DEC-10 Pascal. The programs to be linked must be declared as "EXTERN" in the main program. The main program is compiled as usual. The separately compiled program must have the .REL file present. The execute command then must include the main program filename as well as the separately compiled program filename.

I.4 STACK-FRAME CONTROL DATA

STATIC LINK : The static link records the textual nesting level of the program as it was orginally written. It is used for run-time addressing of variables and objects (Ref 5 : 29).

DYNAMIC LINK : The dynamic link marks the base of the calling procedure's activation record. It is used to deallocate stack space upon completion of the procedure's execution (Ref 5 : 29).

PROGRAM COUNTER : Storage space is provided for the current value of the processor's working registers. This is necessary since a task may have to give up its processor at any time (Ref 5 : 29).

TASK FLAG : The task flag is a Boolean variable that indicates whether or not the stack frame is a task. It is used to indicate task boundaries when processing run-time exceptions raised in the program (Ref 5 : 29).

ACTIVE NESTED TASK COUNTER : The active nested task counter is used to record the number of nested tasks currently active in the given stack frame (Ref 5 : 29-30).

WAITING FLAG : The waiting flag is a boolean variable that indicates whether or not the parent task is waiting to terminate its execution (Ref 5 : 30).

STACK-FRAME CONTROL DATA

EXCEPTIONS : This is not implemented in the Garlington compiler yet. This word will be used to record information on exceptions handled within the block (Ref 5 : 30).

PRIORITY : This word is a run time record of the task's priority (Ref 5 : 30).

TOP OF STACK : The top-of -stack control word provides temporary stroage for the processor's "T" register (Ref 5 : 30).

BASE : The base control word provides temporary storage for the processor's "BASE" register (Ref 5 : 30).

LINK : When a task is entered into a queue, the link control word points to the next task waiting in the queue (Ref 5 : 30).

HEAP : The heap postion provides temporary storage for the processor's "HEAP" register (Ref 5 : 30).

DATA LOCK : This boolean variable indicates whether or not the task frame is currently being accessed by another task (Ref 5 : 30).

CALLER : When a called task executes an accept statement for a particular entry, a pointer to the base of the accepted caller is stored in this word (Ref 5 : 31).

RETURN : This control word is used to record the return

value of the program counter during a procedure call (Ref 5 : 31).

ENTRY : This word records the number of entries declared in the current activation, and is used to compute the amount of space required for entry frame control data (Ref 5 : 31).

I.5 INSTRUCTION SET

RELATIONAL OPERATORS :

EQUAL, GTR, GTREQ, LESS, LESSEQ, ZXOR, ZAND, ZOR, ZNOT

INTEGER ARITHMETIC OPERATORS :

IADD, IDIV, IMULT, INEGATE, ISUB, IMOD, IREM

SINGLE-WORD LOADS AND STORES :

ILOAD, ISTORE, ILOADCONST

TASKING OPERATORS :

CALLENTRY, ACCEPT, RELEASE, TERMINATE, ENTILOAD, ENTISTORE

INPUT/OUTPUT OPERATORS :

SPUT, IPUT, IGET

MISCELLANEOUS INSTRUCTIONS :

CALL, PARAMSHIFT, RETURN, JMP, JMPT, JMPF, INCT

I.6 COMPILER USER'S GUIDE

This appendix describes the input accepted by the test compiler used by the debugger and the output which results. An example program is also included.

Input to the program is an Ada text file whose constructs have been included as part of the implemented subset. The language constructs used to compose input programs are listed below.

1. Integer variables. Number declarations and variable initializations are not implemented.
2. Package declarations.
3. Procedures and functions with parameters.
4. Task declarations.
5. Selected components may be used to open visibility to objects that are within scope but which are not directly visable.
6. Most integer arithmetic or Boolean expressions may be used including those using short circuit conditions. (REM, **, &, IN are not implemented)
7. The following statements may be used:
   a. Assignment
   b. Procedure, function or entry calls
   c. Exit
   d. Return
   e. IF THEN ELSEIF ELSE
   f. Accept
   h. Loops (except FOR loop)

The output of the program is dependent on a specially defined pragma. This pragma was added to allow more direct control of the program throughout its development. Its format is:

PRAGMA TOGGLE (<OPTION_STRING>),

where <OPTION_STRING> is composed of selections from the following list of options: EXECUTE, TRACESTORE, PRINTCODE, TRACEPARSE, TRACETOK, AND DEBUG. Multiple selections must be separated by commas.

All of these options are initially off. To select an option, list it in an option string, and the compiler's output will be as defined below:

EXECUTE; If no errors are detected in the input program, the program will be executed.

TRACESTORE: TRACESTORE will do nothing unless EXECUTE is also selected. If EXECUTE is selected, each value stored during execution of an ISTORE or ENTRISTORE command will be printed.

PRINTCODE: The code generated by the compiler is formatted and printed.

TRACEPARSE: Each transition or reduction made by the parsing automation is printed. This listing is fairly long even for a short program.

TRACETOK: The representation of each token passed from the scanner to the parser is printed. This representation consists of the token's vocabulary index as output from the automatic parser generator.

DEBUG: The code (which is in the form of integer), symbol table, and line number table is written to file COMCOD.TXT. The following example illustrates the effect of selecting the DEBUG option given a simple input program.

CODE

OP	LV	ADDR
23	0	1
22	0	20
13	0	1
14	0	17
13	0	2
14	0	18
13	0	3
14	0	19
40	0	5
37	0	32
37	0	65
37	0	32
37	0	61
37	0	32
12	0	17
39	1	0
40	0	5
37	0	32
37	0	66
37	0	32
37	0	61
37	0	32
12	0	18
39	1	0
40	0	5
37	0	32
37	0	67
37	0	32
37	0	61
37	0	32

```
          12     0        19
          39     1         0
          27     0         0
```

SOURCE CODE

```
 -1 --TEST
 -1 PRAGMA TOGGLE (DEBUG);
 -1
 -1 PROCEDURE MAIN IS
 -1    A, B, C : INTEGER;
  0 BEGIN
  1    A := 1;
  3    B := 2;
  5    C := 3;
  7    PUT (" A = ");   PUT_LINE (A);
 15    PUT (" B = ");   PUT_LINE (B);
 23    PUT (" C = ");   PUT_LINE (C);
 31 END MAIN;
```

SYMBOL TABLE

```
B              1      0     18
A              1      0     17
C              1      0     19
MAIN              3              0
```

I.7 DEBUGGER USER'S GUIDE

This appendix describes the input accepted by the debugger and the output which results by executing the debugger program. Several example debugger options are included.

INPUT: Input to the debugger program is the file 'COMCOD.TXT' which has been generated by the Ada compiler. The specific format of this file can be seen in the Ada Compiler User's Guide. Additional interactive input is required and is dependent on specially defined options. These options are:

B (BREAKPOINTS): Sets the breakpoint at specified source lines.

T (TRACE): Sets trace request.

S (SINGLE STEP): Sets single step execution.

N (MULTI-STEP): Sets multiple step execution.

D (DISPLAY): Displays user specified identifiers.

M (MODIFY VALUE): Modifies user specified identifier value.

O (OUTPUT STACK): Dumps current stack.

J (JUMP): Changes the point of execution.

P (MODIFY PROGRAM): Links to "Dummy" Editor.

C (CONTINUE): Continue execution using previous option.

The following examples illusrate the effect of selecting each of the debugger options and show what additional information is required from the user for the following program.

```
1    --TEST PROGRAM USED FOR DIPLAYING DEBUGGER OPTIONS
2    PRAGMA TOGGLE (DEBUG);
3
4    PROCEDURE MAIN IS
5      A, B, C : INTEGER;
6     BEGIN
7      A := 1;
8      B := 2;
9      C := 3;
10     PUT (" A = ");  PUT_LINE (A);
11     PUT (" B = ");  PUT_LINE (B);
12     PUT (" C = ");  PUT_LINE (C);
13   END MAIN;
```

S SINGLE STEP AND H HELP OPTION

```
*S

  THE PROGRAM IS NOW IN THE SINGLE STEP MODE
  AFTER EVERY SOURCE LINE EXECUTED THE DEBUGGER WILL STOP


  CURRENT LINE IS:
            7    A := 1;
*H

  YOUR OPTIONS FOR THE DEBUG PROGRAM ARE :
  SET A BREAKPOINT, TYPE: B
  TRACE, TYPE: T
  EXECUTE N STATEMENTS AT A TIME, TYPE: N
  SINGLE STEP, TYPE: S
  JUMP TO SOURCE LINE, TYPE: J
  DISPLAY A VALUE, TYPE: D
  MODIFY THE PROGRAM, TYPE: P
  CONTINUE EXECUTION, TYPE: C
  QUIT EXECUTION, TYPE: Q
  MODIFY A VALUE, TYPE: M
  OUTPUT THE STACK,TYPE: O
```

T TRACE

```
*T

 A IS NOW SET AT              1
 B IS NOW SET AT              2
 C IS NOW SET AT              3
 A =                          1
 B =                          2
 C =                          3

 MAIN EXIT
```

D DISPLAY

```
*S

 THE PROGRAM IS NOW IN THE SINGLE STEP MODE
 AFTER EVERY SOURCE LINE EXECUTED THE DEBUGGER WILL STOP


 CURRENT LINE IS:
            7    A := 1;
*C

 CURRENT LINE IS:
            8    B := 2;
*D


 TYPE IN THE IDENTIFIER YOU WISH TO DISPLAY
A EQUALS      1 DECLARED AT FRAME  1

 IF YOU WANT TO DISPLAY ANOTHER IDENTIFIER TYPE: C
 IF NOT TYPE IN ANY CHARACTER

*N


  CURRENT LINE IS:
            9    C := 3;
```

M MODIFY

```
*S

 THE PROGRAM IS NOW IN THE SINGLE STEP MODE
 AFTER EVERY SOURCE LINE EXECUTED THE DEBUGGER WILL STOP

  CURRENT LINE IS:
            7   A := 1;
*C

  CURRENT LINE IS:
            8   B := 2;
*M

 TYPE IN THE IDENTIFIER YOU WISH TO MODIFY
*A

 TYPE IN THE NEW VALUE
*99

 IDENTIFIER MODIFIED

 IF YOU WANT TO MODIFY ANOTHER IDENTIFIER TYPE: C
 IF NOT TYPE IN ANY CHARACTER

*N

 CURRENT LINE IS:
            9   C := 3;

*D

 TYPE IN THE IDENTIFIER YOU WISH TO DISPLAY
*A

A EQUALS     99 DECLARED AT FRAME  1

 IF YOU WANT TO DISPLAY ANOTHER IDENTIFIER TYPE: C
 IF NOT TYPE IN ANY CHARACTER

*N
```

```
CURRENT LINE IS:
          10    PUT (" A = ");  PUT_LINE (A);

A =             99
```

O OUTPUT

```
*S

 THE PROGRAM IS NOW IN THE SINGLE STEP MODE
 AFTER EVERY SOURCE LINE EXECUTED THE DEBUGGER WILL STOP


 CURRENT LINE IS:
            7    A := 1;

*O

              1              0
              2              0
              3              0
              4              1
              5              0
              6              0
              7              0
              8              5
              9              0
             10              0
             11              0
             12              0
             13              0
             14              0
             15              0
             16              0
             17              0
             18              0
             19              0
             20              0


 CURRENT LINE IS:
            8    B := 2;
```

I.8 DEBUGGER SOURCE LISTING

```
PROGRAM DEBUGGER (INPUT:/,OUTPUT);
(* PROGRAM DEBUGGER IS THE DEBUG PROGRAM FOR THE ADA LANGUAGE
WHICH USES THE COMPILER GENERATED BY CAPT ALLAN GARLINGTON.

THE OPTIONS OFFERED BY THIS PROGRAM ARE:
  1. BREAKPOINTS
  2. TRACE
  3. MULTISTEP EXECUTION
  4. SINGLE STEP EXECUTION
  5. MODIFY VALUES OF IDENTIFIERS
  6. DISPLAY VALUES OF IDENTIFIERS
  7. OUTPUT THE STACK

THE PROGRAMS USES THE FOLLOWING PROCEDURES AND FUNCTIONS
  1. FINDBASE : FUNCTION, RETURNS BASE OF STACKFRAME
  2. GO_COMPARE : PROCEDURE, COMPARES INPUT STRING TO
                  SYMBOL TABLE
  3. REINITILIZE : PROCEDURE, RESETS DEBUG COMMANDS
  4. GO_BREAKPOINT : PROCEDURE, SET BREAKPOINTS
  5. GO_TRACE : PROCEDURE, SETS TRACE REQUEST
  6. GO_EXECUTEN : PROCEDURE, SETS MULTISTEP EXECUTION
  7. GO_DISPLAY : PROCEDURE, ALLOWS FOR DISPLAYS OF
                              IDENTIFIERS
  8. GO_MODIFY_PROGRAM : PROCEDURE, LINKS TO "DUMMY" EDITOR
  9. GO_OUTPUT_S : PROCEDURE,OUTPUTS OF STACK
 10. GO_SINGLE_STEP : PROCEDURE, SETS SINGLE STEP OPTION
 11. GO_JUMP : PROCEDURE, ALLOWS FOR JUMPING OVER CODE
 12. CHECK_FOR_BREAK : PROCEDURE, CHECKS FOR BREAKPOINT
                                  CONDITION
 13. CHANGE_VALUE : PROCEDURE, MODIFIES THE IDENTIFIER VALUES
 14. MORE_THAN_ONE : PROCEDURE, DETERMINES WHICH IDENTIFIER
                     (LEVEL) TO BE MODIFIED
 15. GO_MOD_VALUE : PROCEDURE, SET UP FOR MODIFING ID'S VALUE
 16. OPTIONS : PROCEDURE, SETS AND RESETS DEBUGGER OPTIONS
 17. INTERPRET : PROCEDURE, SIMULATES PSEUDO-CODE EXECUTION
 18. INITILIZE : PROCEDURE, SETS FLAGS,
                 READS IN COMPILER GENERATED CODE

INPUT : THE INPUT TO THE DEBUGGER PROGRAM COMES FROM TWO
SOURCES. THE FIRST SOURCE IS THE FILE GENERATE FROM THE
COMPILER. THIS FILE (COMCOD.TXT) HAS THE FOLLOWING
```

```
DEBUGGER SOURCE LISTING

INFORMATION:
  1. PSEUDO-CODE     :     OP LEVEL ADDR
  2. SOURCE-LINE-TABLE   : LINE NUMBERS(CODEX)
  3. SYMBOL-TABLE    :  STRING TYPE LEVEL OFFSET
                        STRING TYPE LEVEL VALUE

  THE SECOND SOURCE OF INPUT COMES FROM THE USER DIRECTLY
 FROM THE SCREEN. THE USER WILL INPUT THE DESIRED OPTIONS
 BASED ON THE PROGRAM PROMPTS.

  ***NOTE ON THE INPUTS- DEC-10 PASCAL USES A BUFFER AREA FOR
  ***CHARACTER INPUT, THEREFORE FOR EVERY CHARACTER READ A
  ***DUMMY READ HAS TO BE PERFORMED TO MANAGE THIS.

OUTPUT : THE OUTPUT FROM THE PROGRAMS ARE WRITTEN TO THE
SCREEN FOR THE USER. THE OUTPUTS ARE EITHER THE PROMPTS FOR
THE USER,THE TRACE OR THE WRITES FROM THE PROGRAM BEING
EXECUTED.

*)



CONST


  (* CONSTANTS USED BY THE COMPILER *)

  CODEXMAX = 2500;          (*LENGTH OF THE CODE ARRAY*)
  LINELENGTH = 120;         (*MAX LENGTH OF AN INPUT LINE*)
  PAGESIZE = 63;            (*NUMBER OF LINES PRINTED / PAGE*)
  MEMORYSIZE = 2037;        (*SIZE OF SIMULATOR'S STACK MEMORY*)
  TASKFRAMESIZE = 17;       (*SIZE OF A PROC/TASK STACK FRAME*)
  ENTRYFRAMESIZE = 3;       (*SIZE OF AN ENTRY'S STACK FRAME*)
  TABSIZE = 64;             (*SIZE OF SYMBOL TABLE LIST*)
  MAXSSTORE = 70;           (*MAX STRING LENGTH OF VARIABLES*)
  NUMPROCESSORS = 3;        (*NUMBER OF SYSTEM PROCESSORS*)
  PRIMARYPROCESSOR = 1;     (*PROCESSOR WHICH INITIZES SYSTEM*)
  EXECUTIONLENGTH = 5;      (*NUM OF INST EXECUTED/TIMESLICE*)

  (* DESCRIPTION OF THE TASK ACTIVATION RECORD *)

  SLINKO = 0; DLINKO = 1; PCO = 2; TASKFLAGO = 3;
  ANTO = 4; WAITO = 5; EXCEPTO = 6; PRIORITYO = 7;
  TOFF = 8; BASEO = 9; LINKO = 10; HEAPO = 11;
  DATALOCKO = 12; CALLERO = 13; RETURNO = 14; ENTRYO = 15;
  SUBO = 16;

  (* DESCRIPTION OF THE ENTRY FRAME *)
```

DEBUGGER SOURCE LISTING

```
  EGATEO = 0; EADDRO = 1; EQUEO = 2;


TYPE

 PRIORITIES = 0..5;



  ADDRESSRANGE = 0..MEMORYSIZE;
  OPERATION = (  (*SEE PROCEDURE INTERPRET FOR A FUNCTIONAL
                  DESCRIPTION OF EACH OPERATOR*)
  (*RELATIONAL OPERATORS-------------------*)
    EQUAL, GTR, GTREQ, LESS, LESSEQ,
    NOTEQ, ZIN, ZNOT, ZXOR, ZAND, ZOR,
  (*INTEGER (SINGLE WORD) OPERATORS--------*)
    ILOAD, ILOADCONST, ISTORE, IADD,
    ISUB, IDIV, IMULT, INEGATE, IMOD, IREM,
  (*REAL OPERATORS-------------------------*)
    RLOAD, RLOADCONST, RSTORE, RADD,
    RSUB, RDIV, RMULT, RNEGATE,
  (*TASKING OPERATORS----------------------*)
    ACTIVATE, CALLENTRY, ACCEPT, RELEASE, TERMINATE,
    ENTILOAD, ENTISTORE, KILLTASK,
  (*I/O OPERATORS--------------------------*)
    IGET, IPUT, SPUT,
  (*MISCELLANEOUS OPERATORS----------------*)
    INCT, JMP, JMPF, JMPT, RAISE, RETURN, ZNEW, CONCAT, EXPON
    CALL, DATA, PARAMSHIFT);


  INSTRUCTION =  RECORD
          (*
          ----------------------------
          ! OP CODE ! LEVEL ! ADDRESS !
          ----------------------------
          *)
                  OP : OPERATION;
                  LEVEL : INTEGER;
                  ADDR  : INTEGER;
                END  (*INSTRUCTION*);


MACHINEDESCRIPTION = RECORD
  (* REGISTERS *)     PC,
                      HEAP,
                      BASE,
                      T : ADDRESSRANGE;
                      IR : INSTRUCTION;
  (* HOUSE KEEPING*) LEXICAL : ADDRESSRANGE;
                      STATE : (BUSY,IDLE);
                      CURRENTJOB : ADDRESSRANGE;
```

```
                    ICOUNT : INTEGER;
                    END; (*MACHINE DESCRIPTION*)


(* IDENTIFIER INFORMATION GENERATED BY THE COMPILER *)
   SYMREC = RECORD
            STRINGLEN : INTEGER; (*LENGTH OF IDENTIFIER*)
            STRINGSTORE : ARRAY[1..MAXSSTORE] OF CHAR;
            TYPES : INTEGER; (*1=VAR,2=CONST,3=PROC*)
            SYLVL : INTEGER; (*LEVEL IDENTIFIER DECLARED*)
            SYOFFSET : INTEGER; (*ADDRESS OFFSET FROM BASE*)
            SYVALUE : INTEGER; (* VALUE FOR CONSTANTS*)
            END; (* SYMREC*)


  KINDS = (SUBP,VARIABLE,PARAM,TYPENAME,LABELNAME,XSTNG,
           STANDARD,PACKAGE,TASK,ENTRY,UNDEFINED);


  CODESOURCE = RECORD
   (* USED TO RELATE SOURCE LINE TO PSEUDO-CODE *)
     CODENUM : INTEGER;
     SOURCECODE : ARRAY [1..50] OF CHAR;
   END;



VAR

  ZDATE : PACKED ARRAY [1..9] OF CHAR;
  ZTIME : INTEGER;


(* CODE GENERATED FROM THE COMPILER*)

  CODE : PACKED ARRAY [0..CODEXMAX] OF INSTRUCTION;
  CODEX : INTEGER;        (*INDEX TO ARRAY 'CODE'.  POINTS TO
                    THE LAST INSTRUCTION ADDED TO THE ARRAY*)

(* ARRAY OF MATCHING ID'S FOR DISPLAY AND MODIFY PROCEDURES*)

  MATCHTABLE : ARRAY[1..TABSIZE] OF INTEGER;

(* LIST FOR THE SYMBOL TABLE *)

  SYMTABLE : ARRAY [1..TABSIZE] OF SYMREC;

(* SAVE VALUE FOR BREAKPOINT GENERATION *)

 PC1 : ADDRESSRANGE;
```

```
DEBUGGER SOURCE LISTING

(* INPUT FILE FROM THE COMPILER *)

 INFILE : TEXT;

(* SOURCE LINE REFERENCE TABLE *)

 LINENUMTABLE : ARRAY [0 .. 100] OF CODESOURCE;

(* BUFFER FOR INTERACTIVE READS OF CHARACTER STRINGS *)

 BUFFER : ARRAY[1..MAXSSTORE] OF CHAR;

(* INDEX FOR SOURCE LINE BEING EXECUTED *)

 SOURCELINECOUNT : INTEGER;

(* FLAGS USED TO STOP EXECUTION FOR CONDITIONS *)
 TERMINATION, FIRSTPASS, TRACE,
 STOPEXECUTION, SINGLESTEP, MULTISTEP : BOOLEAN;
 BREAKPOINT : INTEGER;
 CURRENTSTEP, NUMSTEPS : INTEGER;

(* RUN TIME STACK *)

 S : ARRAY [1..MEMORYSIZE] OF INTEGER;
 READY : ARRAY[PRIORITIES] OF ADDRESSRANGE;
 PROCESSOR : ARRAY[1..NUMPROCESSORS] OF MACHINEDESCRIPTION;
 CURRENTPROCESSOR : INTEGER;
 NEWPTR,TEMPTR,I,EFRAMEPTR,TEMPBASE : INTEGER;



(************************************************)
(* FUNCTION FIND_BASE                           *)
(************************************************)

FUNCTION FINDBASE (LEV, TEMPBASE : INTEGER) : INTEGER;
(* FIND BASE FINDS THE BASE OF THE STACKFRAME *)
BEGIN
 WHILE LEV > 0 DO
 BEGIN
  TEMPBASE := S[TEMPBASE];
  LEV := LEV - 1
 END; (*WHILE*)
 FINDBASE := TEMPBASE
END; (*FINDBASE*)


(************************************************)
(* PROCEDURE GO_COMPARE                         *)
(************************************************)
```

DEBUGGER SOURCE LISTING

```
PROCEDURE GOCOMPARE;

(* COMPARE READS AN INPUT FROM THE TERMINAL
   INTO THE BUFFER AND COMPARES THE STRING
   TO THE SYMBOL TABLE. MORE THAN ONE MATCH
   CAN OCCUR THEREFORE THE MATCHING SYMBOL
   TABLE POINTERS ARE STORED IN THE ARRAY
   MATCH_TABLE FOR ALL MATCHES    *)


VAR
 CH : CHAR;
 I, J, K : INTEGER;
 MATCH : BOOLEAN;

BEGIN
 FOR K := 1 TO TABSIZE DO   (* INITILIZE MATCHTABLE *)
  MATCHTABLE [K] := 0;
 FOR K := 1 TO MAXSSTORE DO (*INITILIZE BUFFER*)
  BUFFER [K] := ' ';
 K := 1;
 READ(CH);                  (*DUMMY READ*)
 READ(BUFFER:I:[' ']);(*READ IN IDENT OR PROCEDURE STRING*)
 FOR J := 1 TO TABSIZE DO (* LOOP THRU SYMBOL TBL FOR MATCH*)
 BEGIN
  WITH SYMTABLE [J] DO
  BEGIN
   FOR I := 1 TO MAXSSTORE DO
    BEGIN
     IF STRINGSTORE [I] <> BUFFER [I] THEN
      BEGIN
       MATCH := FALSE;
       I := MAXSSTORE;
      END
     ELSE
      MATCH := TRUE;
   END; (*FOR*)
   IF MATCH THEN
    BEGIN
    (*MORE THAN ONE IDENTIFIER COULD HAVE SAME STRING *)
     MATCHTABLE [K] := J; (*SAVE IN MATCH TABLE ALL MATCHES*)
     K := K + 1;           (*ON DIFFERENT LEVELS *)
    END;
  END; (*WITH*)
 END; (*FOR*)
END;(* COMPARE *)


(*************************************************************)
(**** PROCEDURE REINITILIZE ******************************)
(*************************************************************)
```

```
(*RESETS THE DEBUG COMMANDS AND VALUES WHEN THE MODE IS TO
  BE CHANGED FROM ONE OPTION TO ANOTHER*)


PROCEDURE REINITILIZE;

BEGIN

 SINGLESTEP := FALSE;
 BREAKPOINT := -2;
 CURRENTSTEP := -2;
 MULTISTEP := FALSE;
END; (* REINITILIZE *)


(*************************************************************
  PROCEDURE GOBREAKPOINT*************************************
*************************************************************

PROCEDURE GOBREAKPOINT;
(* BREAKPOINT : 1. SENDS A MESSAGE TO THE USER
                2. READS IN REQUESTED BREAKPOINT
                3. CHECKS INPUT AS A VALID BREAKPOINT
                4. SETS THE BREAKPOINT *)

VAR
 BK : INTEGER;

BEGIN (* PROC BREAKPOINT *)

 WRITELN('  TO SET A BREAKPONT ENTER SOURCE LINE NUMBER ');
 WRITELN('  SOURCE LINE NUMBER MUST BE AN INTEGER VALUE ');
 WRITELN;

 READ(BK);
 WRITELN;
 IF BK > SOURCELINECOUNT THEN
 BEGIN
  WRITELN('  INPUT CAN NOT BE GREATER THAN :  ',SOURCELINECOU
  WRITELN('  TYPE IN NEW NUMBER ');
  READ(BK);
  WRITELN
 END; (* END IF - NEED ANOTHER CHECK *)

 BREAKPOINT := LINENUMTABLE [BK].CODENUM;

END; (* END GOBREAKPOINT *)


(*************************************************************
```

DEBUGGER SOURCE LISTING

```
(** GO_TRACE*************************************************
(**********************************************************

(*TURNS TRACE FLAG ON AFTER THE REQUEST IS MADE BY THE USER*)


PROCEDURE GOTRACE;

BEGIN

 TRACE := TRUE

END; (* END GOTRACE *)



(**********************************************************
(**** GO_EXECUTEN *****************************************
(**********************************************************

(* EXECUTEN SETS UP THE USFR REQUEST FUR MULTISTEP EXECUTION.
   THIS ALLOWS THE EXECUTION OF "N" LINES OF SOURCE CODE TO
   BE PERFORMED AND THEN A BREAK *)


PROCEDURE GOEXECUTEN;

BEGIN
 MULTISTEP := TRUE;
 CURRENTSTEP := 0;
 WRITELN( ' TO EXECUTE N STATEMENTS AND THEN BREAK ');
 WRITELN(' ENTER THE INTEGER VALUE FOR HE NUMBER OF ');
 WRITELN(' SOURCE LINES TO BE EXECUTED ');
 READ(NUMSTEPS);

END; (*GO_EXECUTEN*)


(**********************************************************)
(**** GO_DISPLAY ******************************************)
(**********************************************************)

(*DISPLAY ALLOWS THE USER TO SEE THE IDENTIFIERS VALUE
  ON THE STACK *)


PROCEDURE GODISPLAY;
VAR
 CH : CHAR;
 I, J, K, L, NUM, TEMPBASE : INTEGER;
 ANOTHER : BOOLEAN;
```

DEBUGGER SOURCE LISTING

```
BEGIN
 ANOTHER := TRUE;
 WHILE ANOTHER DO
 BEGIN
  WRITELN;
  WRITELN(' TYPE IN THE IDENTIFIER YOU WISH TO DISPLAY ');

  GOCOMPARE;

  IF MATCHTABLE [1]  = 0 THEN
   WRITELN(' NO MATCH FOUND FOR THIS IDENTIFIER ')
  ELSE
   BEGIN   (* AT LEAST ONE MATCH HAS BEEN FOUND *)
    FOR I := 1 TO TABSIZE DO
     BEGIN
      IF MATCHTABLE [I] <> 0 THEN   (*COULD BE DEFINED AT
                                     MORE THAN ONE LEVEL*)
       BEGIN
        J := MATCHTABLE [I];
        IF J > 0 THEN
        WITH SYMTABLE [J] DO
         BEGIN
          IF TYPES = 2 THEN  (* IDENTIFIER IS DEFINED
                                AS CONSTANT *)
          BEGIN
           FOR K := 1 TO STRINGLEN DO
            WRITE(STRINGSTORE [K]);
           WRITELN(' = ',SYVALUE);
          END;
          IF TYPES = 1 THEN
                   (* IDENTIFIER IS A VARIABLE INTEGER *)
          BEGIN
           FOR K := 1 TO STRINGLEN DO
                         (*FIND WHICH LEVEL DEFINED AT*)
            WRITE(STRINGSTORE [K]);
           WITH PROCESSOR [CURRENTPROCESSOR], IR DO
           BEGIN
            NUM := LEXICAL;
            TEMPBASE := BASE;
            K := 0;
            WHILE NUM <> 0 DO
             BEGIN
              IF S[TEMPBASE + SUBO] = SYLVL THEN
              BEGIN          (* DISPLAY THE ID'S VALUE *)
               K := K + 1;
               IF K > 1 THEN
               WRITELN('  EQUALS ',S[TEMPBASE + SYOFFSET]:5,
                 ' DECLARED AT FRAME ',NUM:2)
               ELSE
               WRITELN(' EQUALS ',S[TEMPBASE + SYOFFSET]:5,
                 ' DECLARED AT FRAME ',NUM:2);
              END;
```

```
                IF TEMPBASE <> 0 THEN
                 TEMPBASE := S[TEMPBASE + DLINKO];
                NUM := NUM - 1;
              END;(*WHILE*)
             IF K = 0 THEN
                    (*ID IS NOT DEFINED AT ANY FRAME YET*)
              BEGIN  (*PROC MUST NOT HAVE BEEN EXECUTED YET*)
               FOR K := 1 TO TABSIZE DO (*OR HAS BEEN
                                         REMOVED FROM STACK*)
               IF SYMTABLE [K].TYPES = 3 THEN
               IF SYLVL = SYMTABLE [K].SYLVL THEN
               BEGIN
                WRITE('.');
                FOR L := 1 TO SYMTABLE[K].STRINGLEN DO
                 WRITE(SYMTABLE[K].STRINGSTORE[L]);
                WRITELN(' IS NOT DEFINED ');
                K := TABSIZE;
               END;
              END;
            END;(*WITH*)

          END;(*IF TYPE 1*)
        END;(*WITH*)
        END;(*IF*)
      END;(*FOR*)
    END;(*ELSE*)
   WRITELN;
   WRITE(' IF YOU WANT TO DISPLAY ANOTHER');
   WRITELN(' IDENTIFIER TYPE: C ');
   WRITELN(' IF NOT TYPE IN ANY CHARACTER');
   READ(CH); (*DUMMY*)
   READ(CH);
   IF CH <> 'C' THEN ANOTHER := FALSE;
  END;(*WHILE*)
END;



(*************************************************************)
(*** GO_MODIFY_PROGRAM ***************************************)
(*************************************************************)

(* WILL ALLOW FOR A DIRECT LINK TO EDITOR*)
(* NOT IMPLEMENTED YET*)


PROCEDURE GOMODIFYPROGRAM;

BEGIN
 WRITELN(' THIS SECTION IS NOT IMPLEMENTED YET SO ');
 WRITELN(' THE DEBUGGER IS NOW ABOUT TO TERMINATE ');
 WRITELN(' YOU CAN GET THE EDITOR ON YOUR OWN. BYE ');
```

```
END;


(*****************************************************)
(*** GO_SINGLE_STEP*********************************)
(*****************************************************)


(* THIS PROCEDURE SETS A BREAKPOINT FOR EVERY NEW
   SOURCE LINE ENCOUNTERED *)

PROCEDURE GOSINGLESTEP;

BEGIN
 SINGLESTEP := TRUE;
 CURRENTSTEP := LINENUMTABLE[PC1].CODENUM;
 WRITELN;
 WRITELN(' THE PROGRAM IS NOW IN THE SINGLE STEP MODE ');
 WRITE(' AFTER EVERY SOURCE LINE EXECUTED THE');
 WRITELN(' DEBUGGER WILL STOP');
 WRITELN;
END;(*SINGLESTEP*)


(***********************************************************)
(*** GO_JUMP *************************************************)
(***********************************************************)

(*JUMP ALLOWS THE USER TO SKIP OVER CODE WITHOUT EXECUTION
OR TO RETURN TO A GIVEN SOURCE LINE FOR EXECUTION AGAIN.
THIS WILL WORK ONLY WITHIN A GIVEN FRAME OF THE STACK. A
JUMP OUTSIDE THE CURRENT STACK FRAME WILL CAUSE FATAL ERROR*)


PROCEDURE GOJUMP;
VAR
 CH : CHAR;
 NUM : INTEGER;
BEGIN
 WRITELN;
 WRITELN(' *****CAUTION*****');
 WRITELN(' YOU MAY JUMP TO A SOURCE LINE AT THE CURRENT LEVEL
 WRITELN(' A JUMP TO ANY OTHER LEVEL WILL CAUSE AN ABORT');
 WRITELN(' IF YOU WISH TO CONTINUE TYPE: C');
 WRITELN(' IF YOU WANT TO FORGET IT TYPE: ANY CHARACTER');
 READ(CH); READ(CH); (*DUMMY*)
 IF CH = 'C' THEN
  BEGIN
   WRITELN(' TYPE IN THE SOURCE LINE YOU WISH TO JUMP TO');
   READ(NUM);
   IF NUM > CODEXMAX THEN TERMINATION := TRUE
```

```
    ELSE
    WITH PROCESSOR [CURRENTPROCESSOR] DO
    PC := LINENUMTABLE [NUM].CODENUM + 1;
  END;

END; (*JUMP*)

(*****************************************)
(*** CHECK_FOR_BREAK *********************)
(*****************************************)

(* THIS PROCEDURE CHECKS TO SEE IF A BREAKPOINT HAS BEEN
ENCOUNTERED. IF THE BREAKPOINT HAS BEEN HIT PC1 IS SET *)



PROCEDURE CHECKFORBREAK;
VAR
 I : INTEGER;
BEGIN
 FOR I := 1 TO SOURCELINECOUNT DO
 BEGIN
  IF LINENUMTABLE[I].CODENUM = PC1 THEN
   BEGIN
    IF CURRENTSTEP <> PC1 THEN
      BEGIN
       IF SINGLESTEP THEN  (* SINGLE STEP BREAK?*)
        BEGIN
         CURRENTSTEP := PC1;
         BREAKPOINT := PC1;
        END;
       IF MULTISTEP THEN  (* MULTI STEP BREAK?*)
        BEGIN
         CURRENTSTEP := CURRENTSTEP + 1;
                        (*ADD ONE TO STEP COUNT*)
         IF CURRENTSTEP = NUMSTEPS THEN
          BEGIN
           BREAKPOINT := PC1;
           CURRENTSTEP := 0;
          END;
        END;
     END;
   END;
  END;
END;

(*************************************************)
(*** CHANGE_VALUE ************************************)
(*************************************************)

(* CHANGES THE VALUE AT THE GIVE FRAME TO THE NEW
VALUE SPECIFIED BY THE USER*)
```

DEBUGGER SOURCE LISTING

```
PROCEDURE CHANGEVALUE;
VAR
 NUM, LVL, TEMPBASE, K : INTEGER;
BEGIN
 WITH PROCESSOR [CURRENTPROCESSOR] DO
 WITH SYMTABLE [MATCHTABLE [1]] DO
 BEGIN
  IF TYPES = 2 THEN
            (*REQUESTED ID IS DEFINED AS A CONSTANT *)
  BEGIN
   WRITELN;
   WRITELN(' THE IDENIFIER ASKED FOR IS A CONSTANT');
   WRITELN(' YOU ARE NOT ALLOWED TO CHANGE THIS VALUE');
   WRITELN;
  END;
  IF TYPES = 1 THEN  (* IDENTIFIER IS INTEGER *)
  BEGIN
   NUM := LEXICAL;
   TEMPBASE := BASE;
   K := 0;
   WHILE NUM <> 0 DO (*IS IT DEFINED AT MORE THAN ONE FRAME*)
   BEGIN
    IF S[TEMPBASE + SUBO] = SYLVL THEN
     K := K + 1;    (*TEMP VALUE FOR COUNTING FRAMES*)
    IF TEMPBASE <> 0 THEN
     TEMPBASE := S[TEMPBASE + DLINKO];
    NUM := NUM - 1;
   END; (*WHILE*)
   IF K = 0 THEN  (*NO FRAMES ENCOUNTERED FOR THIS ID*)
    WRITELN(' THE IDENTIFIER IS NOT DECLARED YET ');
   IF K = 1 THEN
   BEGIN    (*FRAME WAS FOUND AND AT ONLY ONE LEVEL*)
    WRITELN(' TYPE IN THE NEW VALUE ');
    READ(NUM);
    TEMPBASE := BASE;
    K := LEXICAL;
    WHILE K <> 0 DO
    BEGIN
     IF S[TEMPBASE + SUBO] = SYLVL THEN
      BEGIN
       S[TEMPBASE + SYOFFSET] := NUM;
       WRITELN(' IDENTIFER MODIFIED ');
       K := 0;
      END
     ELSE
     BEGIN
      IF TEMPBASE <> 0 THEN
       TEMPBASE := S[TEMPBASE + DLINKO];
      K := K - 1;
     END;
    END;(*WHILE*)
```

```
    END;(*IF K = 1*)
    IF K > 1 THEN          (*ID DEFINED AT MORE THAN ONE FRAME*)
    BEGIN
     TEMPBASE := BASE;
     WRITELN(' IDENTIFIER DEFINED IN MORE THAN ONE FRAME. ');
     WRITELN(' TYPE IN THE FRAME YOU WANT IT CHANGED IN. ');
     READ(NUM);
     NUM := LEXICAL - NUM;
     IF NUM < 0 THEN NUM := 0; (*DEFAULT TO CURRENT FRAME*)
     WHILE NUM <> 0 DO
      BEGIN
         TEMPBASE := S[TEMPBASE + DLINKO];
          NUM := NUM - 1;
      END;
     IF S[TEMPBASE + SUBO] = SYLVL THEN
     BEGIN
      WRITELN(' TYPE IN THE NEW VALUE ');
      READ(NUM);
      S[TEMPBASE + SYOFFSET] := NUM;
      WRITELN(' IDENTIFIER MODIFIED ');
     END
     ELSE
      WRITELN(' IDENTIFIER NOT DEFINED IN THIS FRAME ');
    END;(*K > 1*)
   END;(*IF TYPE 1*)
   END;(*END WITH*)
 END;(*END CHANGE VALUE*)



(*********************************************************)
(***MORE_THAN_ONE ********************************************)
(*********************************************************)

(* THE SELECTED ID IS DEFINED IN MORE THAN ONE PROCEDURE*)
(* ASKES USER WHICH PROCEDURE FOR ID IS TO BE USED AND
CHECKS TO MAKE SURE IT IS IN THAT PROCEDURE AND THAT THE
PROCEDURE IS IN ACTIVE*)

PROCEDURE MORETHANONE;
VAR
 CH : CHAR;
 I, J, SAVE, LVL : INTEGER;
 MATCH : BOOLEAN;
 SAVID : ARRAY [1 .. MAXSSTORE] OF CHAR;
BEGIN
 WRITELN;
 WRITELN(' YOU HAVE SELECTED AN IDENTIFIED WHICH IS');
 WRITELN(' DEFINED AT MORE THAN ONE PROCEDURE.');
 WRITELN(' TO MODIFY THE IDENTIFIER YOU MUST GIVE');
 WRITE(' THE PROCEDURE FOR THE IDENTIFIER YOU WANT TO');
 WRITELN(' MODIFY.');
```

```
 WRITELN;

   FOR I := 1 TO MAXSSTORE DO
    SAVID [I] := ' ';
   WRITELN(' TYPE IN THE PROCEDURE FOR THIS IDENTIFIER');
   WITH SYMTABLE [MATCHTABLE [1]] DO
                                (*SAVE IDENTIFIER STRING*)
   BEGIN
    FOR I := 1 TO STRINGLEN DO
                          (*SAVE THE REQUESTED ID IN SAVID*)
      SAVID [I] := STRINGSTORE [I];
      END;(*WITH*)
   GOCOMPARE;    (*GO GET INPUT PROCEDURE*)
   IF MATCHTABLE [1] = 0 THEN
    WRITELN(' NO MATCH FOUND FOR THIS PROCEDURE ')
   ELSE
    BEGIN  (*FIND THE MATCHING IDENTIFIER
              AT PROCEDURE LEVEL IN STACK*)
     SAVE := SYMTABLE [MATCHTABLE[1]].SYLVL;
     MATCH := FALSE;
     MATCHTABLE [1] := 0;
     FOR J := 1 TO TABSIZE DO
     BEGIN
      WITH SYMTABLE [J] DO
      BEGIN
       FOR I := 1 TO MAXSSTORE DO
       BEGIN
        IF STRINGSTORE [I] <> SAVID [I] THEN
        BEGIN
         MATCH := FALSE;
         I := MAXSSTORE;
        END
        ELSE
         IF SYLVL = SAVE THEN
          MATCH := TRUE;
       END;(*FOR*)

       IF MATCH THEN
       BEGIN
         MATCHTABLE [1] :=J;
         J := TABSIZE;
       END;
     END;(*WITH*)
    END;(*FOR*)
   END;(*ELSE*)
   IF MATCHTABLE [1] = 0 THEN
     WRITELN(' THIS IDENTIFIER NOT DEFINED IN THE PROCEDURE')
   ELSE
    CHANGEVALUE;
 END;(* MORE THAN ONE*)

(*************************************************************)
```

DEBUGGER SOURCE LISTING

```
(*** GO_MOD_VALUE*********************************************)
(*************************************************************)

(*THE USER HAS REQUESTED TO MODIFY AN ID, A CHECK MUST BE*)
(*PERFORMED TO SEE IF IT IS VALID AND AT HOW MANY LEVELS IT*)
(*IS DEFINDED AT*)


PROCEDURE GOMODVALUE;
VAR
 CH : CHAR;
 NUM : INTEGER;
 ANOTHER : BOOLEAN;
BEGIN
 ANOTHER := TRUE;
 WHILE ANOTHER DO
  BEGIN
   WRITELN;
   WRITELN(' TYPE IN THE IDENTIFIER YOU WISH TO MODIFY');
   GOCOMPARE;
   NUM := 0;
   FOR I := 1 TO TABSIZE DO
    BEGIN
     IF MATCHTABLE [I] <> 0 THEN
      NUM := NUM + 1;
    END;
   IF MATCHTABLE [1] = 0 THEN
    BEGIN
     WRITELN(' NO MACH FOUND FOR INPUT IDENTIFIER');
    END
   ELSE
    BEGIN
     IF NUM > 1 THEN MORETHANONE
     ELSE CHANGEVALUE;
    END;
   WRITELN;
   WRITELN('IF YOU WANT TO MODIFY ANOTHER IDENTIFIER TYPE:C')
   WRITELN(' IF NOT TYPE IN ANY CHARACTER');
   READ(CH);READ(CH);(*DUMMY*)
   IF CH <> 'C' THEN ANOTHER := FALSE;
  END;(*WHILE*)
END;



(*************************************************************)
(***GO_OUTPUT_S ************************************************)
(*************************************************************)

(* OUTPUTS DISPLAYS THE CONTENTS OF THE STACK *)

PROCEDURE GOOUTPUTS;
```

DEBUGGER SOURCE LISTING

```
VAR I : INTEGER;
BEGIN
  WITH PROCESSOR [CURRENTPROCESSOR] DO
  FOR I := 1 TO T DO
   WRITELN(' ',I,' ',S[I]);
END;



(*************************************************************)
(*** OPTIONS *************************************************)
(*************************************************************)


(* OPTIONS READS IN THE REQUESTS FROM THE USER AND CALLS THE
PROCEDURE INVOLVED FOR SET UP *)


PROCEDURE OPTIONS;
VAR
 I, J : INTEGER;
 CH : CHAR;

BEGIN
 STOPEXECUTION := FALSE;
 WRITELN('  PLEASE TYPE IN COMMAND THAT YOU WISH TO PERFORM '
 WRITELN;
 WRITE('  IF YOU WANT A LIST OF THE AVAILABLE COMMANDS');
 WRITELN(' TYPE H FOR HELP ');
 IF NOT FIRSTPASS THEN
  BEGIN
   WRITELN(' CURRENT LINE IS: ');
   FOR I := 1 TO SOURCELINECOUNT DO
    IF LINENUMTABLE [I].CODENUM = (PC1 - 1) THEN
      J := I;
   WRITE(' ',J,' ');
   FOR I := 1 TO 50 DO
     WRITE(LINENUMTABLE [J].SOURCECODE[I]);
   WRITELN;
  END;
 WRITELN; READ(CH);
 READ(CH);
 WRITELN;
 IF CH = 'H' THEN
 BEGIN
  IF FIRSTPASS THEN
  BEGIN
   WRITELN('  YOUR OPTIONS FOR THE DEBUG PROGRAM ARE :');
   WRITELN('  SET A BREAKPOINT, TYPE: B');
   WRITELN('  TRACE, TYPE: T ');
   WRITELN('  EXECUTE N STATEMENTS AT A TIME, TYPE: N');
```

```
   WRITELN('  SINGLE STEP, TYPE: S');
   WRITELN;
   READ(CH);   READ(CH);  (*DUMMY CHAR READ*)
   WRITELN
  END
  ELSE
  BEGIN
   WRITELN('  YOUR OPTIONS FOR THE DEBUG PROGRAM ARE :');
   WRITELN('  SET A BREAKPOINT, TYPE: B');
   WRITELN('  TRACE, TYPE: T');
   WRITELN('  EXECUTE N STATEMENTS AT A TIME, TYPE: N');
   WRITELN('  SINGLE STEP, TYPE: S');
   WRITELN('  JUMP TO SOURCE LINE, TYPE: J');
   WRITELN('  DISPLAY A VALUE, TYPE: D');
   WRITELN('  MODIFY THE PROGRAM, TYPE: P');
   WRITELN('  CONTINUE EXECUTION, TYPE: C');
   WRITELN('  QUIT EXECUTION, TYPE: Q');
   WRITELN('  MODIFY A VALUE, TYPE: M');
   WRITELN('  OUTPUT THE STACK,TYPE: O');
   READ(CH);READ(CH);
   WRITELN
  END  (* END ELSE  FIRSTPASS*)
 END; (* END IF H *)
IF (CH = 'B') OR (CH = 'S') OR (CH = 'N') THEN REINITILIZE;
IF CH = 'Q' THEN TERMINATION := TRUE;
IF CH = 'B' THEN GOBREAKPOINT;
IF CH = 'T' THEN GOTRACE;
IF CH = 'N' THEN GOEXECUTEN;
IF CH = 'S' THEN GOSINGLESTEP;
IF CH = 'J' THEN GOJUMP;
IF CH = 'D' THEN GODISPLAY;
IF CH = 'P' THEN GOMODIFYPROGRAM;
IF CH = 'M' THEN GOMODVALUE;
IF CH = 'O' THEN GOOUTPUTS;

END; (* END PROC OPTIONS *)


(*****************************************************************
(*  PROCEDURE INTERPRET ******************************************
(*****************************************************************



PROCEDURE INTERPRET;
  (*INTERPRETATION OF THE RELATIONAL OPERATORS IS DEPENDENT
    ON THE VALUE OF ORD (FALSE).  PROPER OPERATION REQUIRES
    THE ORD OF FALSE TO EQUAL 0 *)


  PROCEDURE ASSIGN (QNAME : PRIORITIES ; PROCESSORNUM : INTEG
    (*PROCEDURE ASSIGN REMOVES THE GIVEN TASK FROM THE QUEUE
```

```
      WHERE IT WAITS AND INITIALIZES THE GIVEN PROCESSOR WITH
      THE DATA NECESSARY TO BEGIN EXECUTION.  THE TASK IS
      SPECIFIED BY AN INTEGER WHICH POINTS TO THE STACK
      FRAME OF THE TASK.  THE INITIALIZATION DATA IS STORED
      IN THE TASK'S STACK FRAME.*)
    VAR  TASKPTR : INTEGER;  (*PTR TO TASK ACTIVATION RECORD*
  BEGIN
    (*UNLINK THE GIVEN TASK FROM ITS QUEUE*)
    TASKPTR := READY [QNAME];
    READY [QNAME] := S [TASKPTR + LINKO];

    (*INITIALIZE THE PROCESSOR -- THE REQUIRED DATA IS STORED
      THE STACKFRAME POINTED TO BY 'TASKPTR'*)
    WITH PROCESSOR [PROCESSORNUM] DO BEGIN
      PC := S [TASKPTR + PCO];
      BASE := S [TASKPTR + BASEO];
      T := S [TASKPTR + TOFF];
      HEAP := S [TASKPTR + HEAPO];
      STATE := BUSY;
      CURRENTJOB := TASKPTR;
      ICOUNT := 0;
      END  (*WITH PROCESSORNUM*)
  END  (*ASSIGN*);

  PROCEDURE SCHEDULE;
   (*PROCEDURE SCHEDULE ASSIGNS IDLE PROCESSORS TO WAITING
     TASKS. SCHEDULING IS CONITINUED FOR AS LONG AS THERE
     ARE PROCESSORS THAT ARE IDLE AND TASKS THAT ARE WAITING
     IN THE READY QUEUE*)

    FUNCTION WAITINGTASK : PRIORITIES;
      (*FUNCTION WAITING TASK SEARCHES THE READY QUEUES IN
        ORDER OF PRIORITY (WHERE PRIORITY5 IS THE HIGHEST
        AND PRIORITY1 IS THE LOWEST) AND RETURNS THE QUEUE
        NUMBER OF THE TASK WITH THE HIGHEST PRIORITY*)

      VAR MARKER : PRIORITIES;
    BEGIN
      MARKER := 5;  (*BEGIN THE SEARCH WITH THE HIGHEST PRIOR
                      QUEUE*)
      WHILE READY [MARKER] = 0 DO MARKER := MARKER - 1;
      WAITINGTASK := MARKER
    END  (*WAITING_TASK*);

    FUNCTION IDLEPROCESSOR : INTEGER;
      (*FUNCTION IDLE PROCESSOR SEARCHES FOR AN IDLE SYSTEM
        PROCESSOR.  THE FUNCTION RETURNS
        THE NUMBER OF THE FIRST IDLE PROCESSOR IT FINDS*)

    VAR  PINDEX : INTEGER;
         FOUND : BOOLEAN;
    BEGIN
```

```
      FOUND := FALSE; PINDEX := 1;
      WHILE (PINDEX <= NUMPROCESSORS) AND (NOT FOUND) DO
        IF PROCESSOR [PINDEX].STATE = IDLE THEN
          FOUND := TRUE
        ELSE
          PINDEX := PINDEX + 1;
      IF FOUND THEN
        IDLEPROCESSOR := PINDEX
      ELSE
        IDLEPROCESSOR := 0;
    END  (*FUNCTION IDLEPROCESSOR*);

  BEGIN  (*SCHEDULE_TASK*)
    WHILE (IDLEPROCESSOR <> 0) AND (WAITINGTASK <> 0) DO
      ASSIGN (WAITINGTASK, IDLEPROCESSOR)
  END  (*SCHEDULE*);

  FUNCTION NEWP (VAR HEAPTR : INTEGER;NUMWORDS : INTEGER) : I
  BEGIN
    NEWP := HEAPTR + 1;
    HEAPTR := HEAPTR + NUMWORDS
  END  (*NEW*);

PROCEDURE ENTERINQ (TFRAME, PRIORITY : INTEGER);
  (*PROCEDURE ENTERINQ ENTERS TASK 'TFRAME'
    IN THE SPECIFIED READY QUEUE*)
BEGIN
  S[TFRAME + LINKO] := 0;
  IF READY [PRIORITY] <> 0 THEN BEGIN (*TASKS IN QUEUE*)
    TEMPTR := READY [PRIORITY];
    WHILE S[TEMPTR + LINKO] <> 0 DO
      TEMPTR := S[TEMPTR + LINKO];
    (*TEMPTR NOW POINTS TO THE LAST TASK IN THE QUEUE*)
    S [TEMPTR + LINKO] := TFRAME;
    END (*TASKS IN QUEUE*)
  ELSE  (*QUEUE EMPTY*)
    READY [PRIORITY] := TFRAME;
END (*ENTER_IN_QUEUE*);

PROCEDURE RESTOFCASE(CURRENTPROCESSOR : INTEGER;IR : INSTRUCT
BEGIN
  WITH PROCESSOR [CURRENTPROCESSOR], IR DO
  CASE OP OF
ZAND : BEGIN
     T := T - 1;
     S[T] := S[T] + S[T + 1];
     IF S[T] = 2 THEN
       S[T] := ORD (TRUE)
     ELSE
       S[T] := ORD (FALSE)
     END  (*XAND*);
```

```
CALL : BEGIN
     S[T + SLINKO + 1] := FINDBASE (LEVEL,BASE);
     S[T + DLINKO + 1] := BASE;
     S[T + PCO + 1] := 0;
     S[T + 1 + TASKFLAGO] := 0;  (*FALSE*)
     S[T + 1 + ANTO] := 0;
     S[T + 1 + WAITO] := 0;
     S[T + 1 + HEAPO] := 0;
     S[T + 1 + BASEO] := 0;
     S[T + 1 + TOFF] := 0;
     S[T + 1 + LINKO] := 0;
     S[T + 1 + PRIORITYO] := 5; (*FIX*)
     S[T + 1 + CALLERO] := 0;
     S[T + 1 + EXCEPTO] := 0;
     S[T + 1 + DATALOCKO] := 0;
     S[T + 1 + ENTRYO] := 0;
     S[T + 1 + RETURNO] := PC;
     S[T + 1 + SUBO] := ADDR;
     BASE := T + 1;  LEXICAL := LEXICAL + 1; PC := ADDR;

     WRITE(' ');
     FOR I := 1 TO TABSIZE DO
     WITH SYMTABLE [I] DO
     BEGIN
      IF S[BASE + SUBO] = SYLVL THEN
       BEGIN
        IF TYPES = 3 THEN
         BEGIN
          FOR I := 1 TO STRINGLEN DO
           WRITE(STRINGSTORE [I]);
           I := TABSIZE;
         END;
       END;
      END;
    WRITELN(' ENTERED');
   END;



PARAMSHIFT :  BEGIN  (*PARAMSHIFT, # OF WORDS, T-INCREMENT*)
  T := T + IR.ADDR;
  FOR I := 1 TO IR.LEVEL DO
    S[T - (I-1)] := S[(T-IR.ADDR) - (I-1)];
  END  (*PARAMSHIFT*);

CONCAT :     ;

EQUAL : BEGIN  T := T - 1;  S[T] := ORD (S[T] = S[T + 1])
        END;

(*EXPON : BEGIN  T := T - 1;  S[T] := S[T] ** S[T + 1]
      END;  MODIFY FOR DIFFERENT ARGUMENT TYPES*)
```

AD-A115 636 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH F/G 9/2
ANALYSIS AND DESIGN OF INTERACTIVE DEBUGGING FOR THE ADA PROGRA--ETC(U)
NOV 81 R L GAUDINO
UNCLASSIFIED AFIT/GCS/MA/81D-3 NL

END
DATE FILMED
7-82
DTIC

DEBUGGER SOURCE LISTING

```
GTR : BEGIN  T := T - 1 ;  S[T] := ORD (S[T] > S[T + 1])
      END;

GTREQ : BEGIN  T := T - 1;  S[T] := ORD (S[T] >= S[T + 1])
        END;

IADD : BEGIN  T := T - 1;  S[T] := S[T] + S[T + 1]
       END;

IDIV : BEGIN  T := T - 1;  S[T] := S[T] DIV S[T + 1]
     END;

IMULT : BEGIN  T := T - 1;  S[T] := S[T] * S[T + 1]
      END;

INEGATE :   S[T] := -S[T];

ISUB : BEGIN  T := T - 1;  S[T] := S[T] - S[T + 1]
     END;

ZIN : BEGIN
    T := T - 2;
    IF (S[T] >= S[T + 1]) AND (S[T] <=S[T + 2]) THEN
      S[T] := ORD (TRUE)
    ELSE
      S[T] := ORD (FALSE)
    END  (*XIN*);

INCT : T := T + ADDR;

JMP : PC := ADDR;

JMPF : BEGIN
       IF S [T] = 0 THEN PC := ADDR;
       T := T - 1
       END;

JMPT : BEGIN
       IF S [T] = 0 THEN PC := ADDR;
       T := T - 1
       END;

LESS : BEGIN  T := T - 1;  S[T] := ORD (S[T] < S[T + 1])
      END;

LESSEQ : BEGIN  T := T - 1;  S[T] := ORD (S[T] <= S[T + 1])
       END;

ILOAD : BEGIN  T := T + 1;  S[T] :=
                S [FINDBASE (LEVEL,BASE) + ADDR]
       END;
```

```
ENTILOAD :  BEGIN
  (*SET TEMPTR TO THE BASE OF CALLER'S TASKFRAME*)
  TEMPTR := S [CURRENTJOB + CALLERO];
  (*SET TEMPTR TO THE TOP OF THE CALLER'S STACK*)
  TEMPTR := S [TEMPTR + TOFF];
  T := T + 1;
  S[T] := S[TEMPTR + IR.ADDR];
  END  (*ENTILOAD*);

ENTISTORE : BEGIN
  (*SET TEMPTR TO THE BASE OF THE CALLER'S TASKFRAME*)
  TEMPTR := S[CURRENTJOB + CALLERO];
  (*SET TEMPTR TO THE TOP OF THE CALLER'S STACK*)
  TEMPTR := S[TEMPTR + TOFF];
  S[TEMPTR + IR.ADDR] := S[T];
  T := T - 1;
  END (*ENTISTORE*);

ILOADCONST : BEGIN T := T + 1;  S[T] := ADDR
       END;

RLOADCONST : (*DEFINE REALS*);

IMOD : BEGIN  T := T - 1;  S[T] := S[T] MOD S[T + 1]
     END;


NOTEQ : BEGIN  T := T - 1;  S[T] := ORD ( S[T] <> S[T] + 1)
        END;
RADD : (*DEFINE FLOATING POINT FORMAT*);

RAISE : (*FILL IN THE BLANK*) ;

RDIV : (*DEFINE FLOATING POINT FORMAT*);

IREM : (*WRITE ROUTINE*);

RETURN : BEGIN
  (*SEE IF THERE ARE ANY ACTIVE NESTED TASKS*)
  IF S [BASE + ANTO] = 0 THEN BEGIN  (*NONE ACTIVE*)
    WRITE(' ');
    FOR I := 1 TO TABSIZE DO
    WITH SYMTABLE [I] DO
     BEGIN
     IF S[BASE + SUBO] = SYLVL THEN
      BEGIN
       IF TYPES = 3 THEN
        BEGIN
         FOR I := 1 TO STRINGLEN DO
           WRITE(STRINGSTORE[I]);
           I := TABSIZE;
```

```
        END;
      END;
     END;
    WRITELN(' EXIT');
    T := BASE - 1;
    PC := S [BASE + RETURNO];
    BASE := S [BASE + DLINKO];
    LEXICAL := LEXICAL - 1;
    END (*NONE ACTIVE*)
  ELSE BEGIN  (*WAIT FOR TASK(S) TO COMPLETE*)
    (*SET PARENT WAITING FLAG*)
    WRITE(' ');
    FOR I := 1 TO TABSIZE DO
    WITH SYMTABLE [I] DO
     BEGIN
      IF S[BASE + SUBO] = SYLVL THEN
       BEGIN
        IF TYPES = 3 THEN
         BEGIN
          FOR I := 1 TO STRINGLEN DO
           WRITE(STRINGSTORE[I]);
          I := TABSIZE;
         END;
       END;
     END;
    WRITE(' EXIT');
    S [BASE + WAITO] := 1;
    (*SET PC BACK TO THE RETURN INSTRUCTION*)
    S [BASE + PCO] := PC - 1;
    S [BASE + HEAPO] := HEAP;
    S [BASE + BASEO] := BASE;
    S [BASE + TOFF] := T;
    LEXICAL := LEXICAL - 1;
    (*GO TO SLEEP*)
    PROCESSOR [CURRENTPROCESSOR].STATE := IDLE;
    ICOUNT := ICOUNT + EXECUTIONLENGTH;
    SCHEDULE;
    END (*WAIT*)
  END (*RETURN*);

RMULT : (*DEFINE FLOATING POINT FORMAT

RNEG  :             ""

RSUB  :             ""          *);


ISTORE : BEGIN
      S [FINDBASE(LEVEL,BASE) + ADDR] := S[T];
      IF TRACE THEN
        BEGIN
         WRITE(' ');
```

```
          FOR I := 1 TO TABSIZE DO
          WITH SYMTABLE [I] DO
           BEGIN
            IF ADDR = SYOFFSET THEN
             BEGIN
              IF SYLVL = S[FINDBASE(LEVEL,BASE) + SUBO] THEN
               BEGIN
                FOR I := 1 TO STRINGLEN DO
                 WRITE(STRINGSTORE [I]);
                 I := TABSIZE;
               END;
             END;
           END;
         WRITELN (' IS NOW SET AT ', S[T]);
        END;
       T := T - 1;
       END;

ZNEW : BEGIN T := T + 1;  S[T] := NEWP (HEAP,ADDR)
     END;

ZNOT : BEGIN
     IF S[T] = 0 THEN
       S[T] := ORD (TRUE)
     ELSE
       S[T] := ORD (FALSE)
     END  (*ZNOT*);

ZOR :  BEGIN
    T := T - 1;
    S[T] := S[T] + S[T + 1];
    IF S[T] = 0 THEN
      S[T] := ORD (FALSE)
    ELSE
      S[T] := ORD (TRUE)
    END  (*ZOR*);

ZXOR : BEGIN
     T := T - 1;
     S[T] := S[T] + S[T + 1];
     IF S[T] = 1 THEN
       S[T] := ORD(TRUE)
     ELSE
       S[T] := ORD (FALSE)
     END  (*ZXOR*);
  END  (*CASE*);
END  (*REST OF CASE*);


BEGIN  (*INTERPRET*)
  (*SET STATE OF ALL PROCESSORS TO IDLE*)
  FOR I := 1 TO NUMPROCESSORS DO PROCESSOR [I].STATE := IDLE;
```

```
  (*INITIALIZE PRIMARY PROCESSOR*)
  WITH PROCESSOR [PRIMARYPROCESSOR] DO BEGIN
    T := 0;
    BASE := 1;
    LEXICAL := 1;
    PC := 0;
    HEAP := MEMORYSIZE;
    STATE := BUSY;
    ICOUNT := 0;
    CURRENTJOB := 1;
    (*INITIALIZE THE STACKFRAME*)
    FOR I := 1 TO TASKFRAMESIZE  DO
      S[I] := 0;
    S[BASE + TASKFLAGO] := 1;  (*FLAG SET*)
    S[BASE + PRIORITYO] := 5;  (*FIX*)
    END  (*WITH PRIMARY PROCESSOR*);
  CURRENTPROCESSOR := PRIMARYPROCESSOR;


  (*INITIALIZE READY QUEUE*)
  FOR I := 1 TO 5 DO READY [I] := 0;
  READY [0] := 1;

  (*BEGIN SIMULATION - PRIMARY PROCESSOR ACTIVE*)
  TERMINATION := FALSE;
  WHILE NOT TERMINATION DO BEGIN
      WITH PROCESSOR [CURRENTPROCESSOR], IR DO
       REPEAT
          IF STOPEXECUTION THEN OPTIONS;
          IR := CODE [PC];
          IF SINGLESTEP THEN CHECKFORBREAK;
          IF MULTISTEP THEN CHECKFORBREAK;
          IF BREAKPOINT = PC THEN STOPEXECUTION := TRUE;
          PC := PC + 1;
          PC1 := PC;
          ICOUNT := ICOUNT + 1;
          CASE OP OF

(* I/O OPERATIONS *)

IGET : BEGIN
  READ (S[T+1]);
  S[FINDBASE (LEVEL,BASE) + ADDR] := S[T+1];
  END  (*IGET*);

IPUT : BEGIN  (*PRINT VALUE ON TOP OF THE STACK*)
  IF LEVEL = 0 THEN (*PUT WITHOUT CARRIAGE RETURN*)
    WRITE (S[T])
  ELSE
    WRITELN (S[T]);
  T := T - 1;
```

```
  END  (*IPUT*);

SPUT : BEGIN  (*PRINT STRING -- LENGTH STORED IN ADDR FIELD*)
  FOR I := 1 TO ADDR DO BEGIN
    WRITE (CHR (CODE[PC].ADDR):1);
    PC := PC + 1;
    END (*FOR*);
  IF LEVEL = 1 THEN (*PUT WITH CARRIAGE RETURN*)
    WRITELN;
  END  (*SPUT*);

ACCEPT :  BEGIN  (*ACCEPT, 0, ENTRY # *)
  (*THIS INSTRUCTION WAS WRITTEN WITH THE ASSUMPTION THAT
    ENTRYFRAMES IMMEDIATELY FOLLOW THE ENTRY OFFSET IN THE
    STACK FRAME.  I.E. THE FIRST ENTRY FRAME BEGINS AT
    BASE + ENTRYO + 1*)
  (*COMPUTE A POINTER TO THE TASK'S 1ST ENTRY FRAME*)
  EFRAMEPTR := CURRENTJOB + ENTRYO + 1 +
               (IR.ADDR - 1)*ENTRYFRAMESIZE;
  (*SEE IF THIS ENTRY HAS ANY TASKS WAITING*)
  IF S[EFRAMEPTR + EQUEO] = 0 THEN BEGIN (*NO TASKS WAITING*)
    S[EFRAMEPTR + EGATEO] := 1;          (*OPEN GATE*)

    (*GO TO SLEEP*)
    S [CURRENTJOB + PCO] := PC - 1; (*POINTS TO THE ACCEPT
                                      INSTRUCTION*)
    S [CURRENTJOB + HEAPO] := HEAP;
    S [CURRENTJOB + BASEO] := BASE;
    S [CURRENTJOB + TOFF] := T;
    (*RELEASE PROCESSOR*)
    PROCESSOR [CURRENTPROCESSOR].STATE := IDLE;
    ICOUNT := ICOUNT + EXECUTIONLENGTH;
    SCHEDULE;
          (*A NEW PROCESSOR IS SELECTED ON EXIT FROM LOOP*)
    END (*NO TASKS WAITING*)
  ELSE BEGIN (*WAITING TASK*)
    (*REMOVE TASK FROM QUEUE*)
    S[CURRENTJOB + CALLERO] := S[EFRAMEPTR + EQUEO];
    S[EFRAMEPTR + EQUEO] := S[S[BASE + CALLERO] + LINKO];
    END (*WAITING TASK*);
  END (*ACCEPT*);

ACTIVATE :  BEGIN  (*ACTIVATE, 0, TASKPTR);
                     (DATA, PRIORITY, HEAP);
                     (DATA, 0, PC);  (DATA, #ENTRIES, T*)
  (*'TASKPTR' IS RELATIVE TO THE BASE OF THE PARENT.
     HEAP AND T ARE RELATIVE TO 'TASKPTR'.  THIS INSTRUCTION
     WAS WRITTEN WITH THE ASSUMPTION THAT ENTRY FRAMES
     IMMEDIATELY FOLLOW THE ENTRY OFFSET
     IN THE STACK FRAME. I.E. THE FIRST ENTRY FRAME BEGINS AT
     BASE + ENTRO + 1*)
```

```
  REPEAT
    (*INITIALIZE THE STACKFRAME*)
    (*NOTE : THE ACTIVATE INSTRUCTION REMAINS IN THE IR
      THROUGHOUT THE INSTRUCTION'S
      EXECUTION, EVEN THOUGH PC IS INCREMENTED*)
    (*SET 'TEMPTR' TO THE BASE OF THE TASK BEING ACTIVATED*)
    TEMPTR := BASE + IR.ADDR;

    S [TEMPTR + SLINKO] :=  BASE;  (*STATIC LINK ASSIGNMENT*)
    S [TEMPTR + DLINKO] := BASE;  (*DYNAMIC LINK ASSIGNMENT*)
    S [TEMPTR + PRIORITYO] := CODE [PC].LEVEL;  (*PRIORITY*)
    S [TEMPTR + BASEO] := TEMPTR;       (*BASE*)
    S [TEMPTR + HEAPO] := TEMPTR + CODE [PC].ADDR;  (*HEAP PT

    PC := PC + 1;
    S [TEMPTR + PCO] := CODE [PC].ADDR;    (*PROGRAM COUNTER*
    PC := PC + 1;
    S [TEMPTR + TOFF] := TEMPTR + CODE [PC].ADDR;(*STACK PTR*
    S [TEMPTR + ENTRYO] :=  CODE [PC].LEVEL;  (* # NUMBER OF
                                          ENTRIES IN TASK*)

    (*CLOSE ALL ENTRIES*)
    FOR I := TEMPTR + ENTRYO + 1 TO TEMPTR +
             ENTRYO + ENTRYFRAMESIZE*CODE[PC].LEVEL DO
      S [I] := 0;  (*CLOSED*)
    PC := PC + 1;  (*PC NOW POINTS PAST THE LAST DATA WORD
                     OF THE ACTIVATE INSTRUCTION*)
 (*INITIALIZE MISCELLANEOUS CONTROL WORDS IN THE STACKFRAME*)
    S [TEMPTR + ANTO] := 0;  (*ACTIVE NESTED TASKS*)
    S [TEMPTR + LINKO] := 0; (*LINK*)
    S [TEMPTR + DATALOCKO] := 0;  (*UNLOCKED*)
    S [TEMPTR + EXCEPTO] := 0;
    S [TEMPTR + TASKFLAGO] := 1;
    S [TEMPTR + WAITO] := 0;
    S [TEMPTR + CALLERO] := 0;
    S [TEMPTR + RETURNO] := 0;

  (*INCREMENT THE ACTIVE NESTED TASK COUNTER OF THE PARENT --
      NOTE: PARENT EXECUTES THIS INSTRUCTION*)
    S [BASE + ANTO] := S [BASE + ANTO] + 1;

    ENTERINQ (TEMPTR, S [TEMPTR + PRIORITYO]);
    IR := CODE [PC];
    IF IR.OP = ACTIVATE THEN PC := PC + 1;
  UNTIL IR.OP <> ACTIVATE;
  (*UNLOCK QUEUES*)
  SCHEDULE;  (*GO TO SLEEP??*)
  END  (*ACTIVATE*);

CALLENTRY : BEGIN  (*CALLENTRY, LEX DIFF, TASKPTR);
                    (DATA, ENTRY#, #ENTRIES*)
  (*THIS PROCEDURE WAS WRITTEN WITH THE ASSUMPTION THAT ENTRY
```

```
  FRAMES IMMEDIATELY FOLLOW ENTRYO IN THE STACK FRAME.
  I.E. FIRST ENTRY FRAME MUST BEGIN AT BASE + ENTRYO + 1*)

  (*'TASKPTR' IS RELATIVE TO THE BASE OF THE PARENT--
  COMPUTE AN ABSOLUTE ADDRESS FOR THE BASE OF THE CALLED TASK
  TEMPBASE (*TASK'S ABSOLUTE ADDRESS*) := FINDBASE (LEVEL, BA
                                          IR.ADDR;

  (*LINK CALLER TO THE CALLED TASK'S ENTRY QUEUE*)
  (*ZERO CALLER'S LINK FIELD*)
  S[CURRENTJOB + LINKO] := 0;
  (*COMPUTE A POINTER TO THE CALLED ENTRY'S QUEUE*)
  EFRAMEPTR := TEMPBASE + ENTRYO + 1 + (CODE[PC].LEVEL - 1)*
               ENTRYFRAMESIZE;
  TEMPTR := EFRAMEPTR + EQUEO;  (*TEMPTR NOW POINTS TO THE
                                 HEAD OF THE ENTRY QUEUE*)
  IF S[TEMPTR] = 0 THEN (*QUEUE EMPTY*)
    S[TEMPTR] := CURRENTJOB    (*LINK CALLER TO THE QUEUE*)
  ELSE BEGIN (*TASKS IN QUEUE*)
    (*FIND THE END OF THE QUEUE*)
    TEMPTR := S[TEMPTR];  (*SET TEMPTR TO THE FIRST TASK IN
                           THE QUEUE*)
    WHILE S[TEMPTR + LINKO] <> 0 DO
      TEMPTR := S [TEMPTR + LINKO];
    (*TEMPTR POINTS TO THE LAST TASK IN QUEUE*)
    S [TEMPTR + LINKO] := CURRENTJOB;(*LINK CALLER TO QUEUE*)
    END (*TASKS IN QUEUE*);
  (*CHECK TO SEE IF THE CALLED TASK'S ENTRY IS OPEN*)
  IF S [EFRAMEPTR + EGATEO] = 1 THEN BEGIN (*ENTRY OPEN*)
    I := TEMPBASE + ENTRYO + 1;
    WHILE I <= TEMPBASE + ENTRYO + CODE[PC].ADDR * ENTRYFRAME
    DO BEGIN
      S [I + EGATEO] := 0;  (*CLOSED*)
      I := I + ENTRYFRAMESIZE;
      END (*WHILE*);
    ENTERINQ (TEMPBASE, S[TEMPBASE + PRIORITYO]);
    END;
  (*GO TO SLEEP -- SAVE THE MACHINE'S REGISTERS IN
    CALLER'S TASKFRAME*)
  S [CURRENTJOB + PCO] := PC + 1;
  S [CURRENTJOB + HEAPO] := HEAP;
  S [CURRENTJOB + BASEO] := BASE;
  S [CURRENTJOB + TOFF] := T;

  (*RELEASE PROCESSOR*)
  PROCESSOR [CURRENTPROCESSOR].STATE := IDLE;
  ICOUNT := ICOUNT + EXECUTIONLENGTH;

  SCHEDULE;  (*DATA LOCK*)
  END  (*CALL ENTRY*);

RELEASE : BEGIN  (*RELEASE, 0, ENTRY# *)
```

```
  (*RETRIEVE THE POINTER TO THE BASE OF THE CALLING TASK'S
    TASKFRAME*)
  TEMPTR := S [CURRENTJOB + CALLERO];
  ENTERINQ (TEMPTR, S[TEMPTR + PRIORITYO]);
  SCHEDULE;

  (*GIVE UP PROCESSOR ??  DATA LOCKS*)
  END (*RELEASE*);

TERMINATE : BEGIN  (*TERMINATE,0,0*)
  TEMPTR := FINDBASE (1, BASE);  (*SET TEMPTR TO THE PARENT'S
                                   TASK FRAME*)
  S [TEMPTR + ANTO] := S [TEMPTR + ANTO] - 1;
  IF (S [TEMPTR + ANTO] = 0) AND
     (S [TEMPTR + WAITO] = 1) THEN
     (*PARENT WAITING TO TERMINATE*)
     (*WAKE UP PARENT*)
     ENTERINQ (TEMPTR, S[TEMPTR + PRIORITYO]);

  (*TERMINATE SELF*)
  PROCESSOR [CURRENTPROCESSOR].STATE := IDLE;
  ICOUNT := ICOUNT + EXECUTIONLENGTH;
  SCHEDULE;
  END (*TERMINATE*);

KILLTASK : BEGIN  (*KILLTASK, 0, TASKPTR*)
  (*** NOTE :  THIS INSTRUCTION HAS NOT BEEN TESTED ***)
  (*COMPUTE AN ABSOLUTE ADDRESS FOR THE CALLED
                                TASK'S STACKFRAME*)
  TEMPBASE (*TASK'S ABSOLUTE ADDRESS*) :=
                           FINDBASE (LEVEL,BASE) + IR.ADDR;
  (*FIND THE PROCESSOR EXECUTING THE TASKTO BE KILLED*)
  TEMPTR := 0;
  FOR I := 1 TO NUMPROCESSORS DO
    IF PROCESSOR [I].CURRENTJOB = TEMPBASE THEN
      TEMPTR := I;
  IF TEMPTR <> 0 THEN BEGIN (*FOUND*)
    (*DECREMENT ANT COUNTER OF PARENT*)
    I := FINDBASE (1,IR.ADDR);
    S [I + ANTO] := S[I + ANTO] - 1;
    (*TELL PROCESSOR TO STOP*)
    PROCESSOR [TEMPTR].STATE := IDLE;
    PROCESSOR [TEMPTR].CURRENTJOB := 0;
    SCHEDULE;
    END (*FOUND*)
  ELSE
    (*LOOK IN THE READY QUEUES, AND REMOVE IF FOUND*);
  END  (*KILL_TASK*);

OTHERS : RESTOFCASE(CURRENTPROCESSOR,IR);

         END  (*CASE*)
```

```
        UNTIL (PC = 0) OR (ICOUNT >= EXECUTIONLENGTH);
        IF PROCESSOR [CURRENTPROCESSOR].PC = 0 THEN
                          (*EXECUTION COMPLETE*)
          TERMINATION := TRUE
        ELSE BEGIN (*SELECT THE NEXT PROCESSOR*)
          CURRENTPROCESSOR := CURRENTPROCESSOR
                              MOD NUMPROCESSORS + 1;
          I := 0;
          WHILE (PROCESSOR [CURRENTPROCESSOR].STATE = IDLE) A
                (I <= NUMPROCESSORS) DO BEGIN
            CURRENTPROCESSOR := CURRENTPROCESSOR
                                MOD NUMPROCESSORS + 1;
            I := I + 1;
            END (*WHILE*);
          IF I > NUMPROCESSORS THEN BEGIN (*ERROR ABORT*)
            TERMINATION := TRUE;
            WRITELN (' LOOPING IN INTERPRET -- ALL PROCESSORS
            END (*ERROR ABORT*)
          ELSE
            PROCESSOR [CURRENTPROCESSOR].ICOUNT := 0;
          END  (*SELECT*)
    END  (*WHILE NOT TERMINATION*)
END  (*INTERPRET*);



(*************************************************************)
(*** INITILIZE ***********************************************
*************************************************************)


(* SETS UP THE FLAGS AND READS IN THE INFORMATION IN
COMCOD.TXT WHICH WAS GENERATED FROM THE COMPILER  *)


PROCEDURE INITIALIZE;
VAR OPX,I, J, LX, AX : INTEGER;
    CH : CHAR;
    DONE : BOOLEAN;

BEGIN
  RESET(INFILE,'COMCOD.TXT');
  FIRSTPASS := TRUE;
  TRACE := FALSE;
  BREAKPOINT := -2;
  CURRENTSTEP := -2;
  MULTISTEP := FALSE;
  SINGLESTEP := FALSE;
  READ(INFILE,CODEX);
  READLN(INFILE);
  FOR I := 0 TO CODEX DO   (* READ IN THE CODE PORTION*)
```

```
   WITH CODE [I] DO
   BEGIN
    READ(INFILE,OPX);
    CASE OPX OF
1 :  OP := EQUAL; 2 :  OP := GTR;     3 :  OP := GTREQ;
4 :  OP := LESS;  5 :  OP := LESSEQ; 6 :  OP := NOTEQ;
7 :  OP := ZIN;   8 :  OP := ZNOT;   9 :  OP := ZNOT;
10 : OP := ZAND;  11 : OP := ZOR;   12 : OP := ILOAD;
13 : OP := ILOADCONST;
14 : OP := ISTORE; 15 : OP := IADD; 16 : OP := ISUB;
17 : OP := IDIV;   18 : OP := IMULT;
19 : OP := INEGATE;20 : OP := IMOD;
21 : OP := IREM;   22 : OP := INCT;
23 : OP := JMP;    24 : OP := JMPF;
25 : OP := JMPT;   26 : OP := RAISE;
27 : OP := RETURN; 28 : OP := ZNEW;
29 : OP := CONCAT; 30 : OP := EXPON;
31 : OP := CALL;   32 : OP := ACTIVATE;
33 : OP := CALLENTRY; 34 : OP := TERMINATE;
35 : OP := ACCEPT;    36 : OP := RELEASE;
37 : OP := DATA;      38 : OP := IGET;
39 : OP := IPUT;      40 : OP := SPUT;
41 : OP := ENTILOAD;  42 : OP := KILLTASK;
43 : OP := ENTISTORE; 44 : OP := PARAMSHIFT;
END; (*CASE*)


    READ(INFILE,LX);
    LEVEL := LX;
    READ(INFILE,AX);
    ADDR := AX;
   END;  (* END WITH *)



    READLN(INFILE);   (* READ IN THE SOURCE LINE TABLE *)
    READ(INFILE,SOURCELINECOUNT);
    READLN(INFILE);
    FOR I := 1 TO SOURCELINECOUNT DO
     BEGIN
      READ(INFILE,LINENUMTABLE [I].CODENUM);
      FOR J := 1 TO 50 DO
       READ(INFILE,LINENUMTABLE [I].SOURCECODE [J]);
      READLN(INFILE)
     END;



  FOR I := 1 TO TABSIZE DO  (*READ IN THE SYMBOL TABLE INFO*)
  WITH SYMTABLE [I] DO
   BEGIN
    STRINGLEN := 0;
```

```
      TYPES := 0;
      SYLVL := 0;
      SYOFFSET := 0;
      SYVALUE := 0;
      FOR J := 1 TO MAXSSTORE DO
       STRINGSTORE [J] := ' ';
     END;
    DONE := FALSE;
    I := 1;
    READ(INFILE,J); (*READ STRINGLEN*)
    WHILE NOT DONE DO
    BEGIN
     WITH SYMTABLE [I] DO
     BEGIN (*WITH*)
      STRINGLEN := J;
      READ(INFILE,CH);
      FOR J := 1 TO STRINGLEN DO
       BEGIN
        READ(INFILE,CH);
        STRINGSTORE [J] := CH
       END; (*END FOR *)
      READ(INFILE,TYPES);
      IF TYPES = 1 THEN
       BEGIN
        READ(INFILE,SYLVL);
        READ(INFILE,SYOFFSET);
       END;
      IF TYPES = 2 THEN
       READ(INFILE,SYVALUE);
      IF TYPES = 3 THEN
       READ(INFILE,SYLVL);
      READ(INFILE,J);

      I := I + 1;
      IF EOF THEN DONE := TRUE;
      IF J = 0 THEN DONE := TRUE;
      IF I = TABSIZE THEN DONE := TRUE;
     END;(* WITH*)
    END; (*WHILE*)
    FOR I := 1 TO (TABSIZE - 1) DO
    FOR J := (I + 1) TO TABSIZE DO
    BEGIN
     IF SYMTABLE [I].TYPES = SYMTABLE [J].TYPES THEN
     IF SYMTABLE [I].SYLVL = SYMTABLE [J].SYLVL THEN
     IF SYMTABLE [I].SYOFFSET = SYMTABLE [J].SYOFFSET THEN
      SYMTABLE [J].STRINGSTORE [1] := ' ';
    END;
    CLOSE(INFILE)

END;  (* END INITIALIZE*)
BEGIN
  INITIALIZE;
```

```
    OPTIONS;
    FIRSTPASS := FALSE;
    INTERPRET;

END.
```

VITA

Richard L. Gaudino was born on 14 May 1951 in DuBois, Pennsylvania to Alfred P. Gaudino and Mary R. (Hetrick) Gaudino. He attended high school at Haverling Central in Bath, New York and graduated in 1969. In September of that year, he entered Corning Community College in Corning, New York and graduated with an Associates of Science in Mathematics. He entered the Air Force in June of 1971. He graduated from the University of Southwestern Louisiana with a Bachelor of Science Degree in Mathematics in May of 1974. He was commissioned in the USAF in August of 1974. His assignments as an officer have included the 3900 Computer Service Squadron, HQ SAC, at Offutt AFB, Nebraska and the Rome Air Development Center at Rome, New York. He then entered the Air Force Institute of Technology School of Engineering at Wright Patterson AFB, Ohio in June of 1980.

Capt Gaudino was married on 15 May 1980 in Bath, New York to Lauren C. Gaudino. They have two sons, Richard and Brian.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/MA/81D-3	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ANALYSIS AND DESIGN OF INTERACTIVE DEBUGGING FOR THE ADA PROGRAMMING SUPPORT ENVIRONMENT		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard L. Gaudino Capt USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE December 1981 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17
LYNN E. WOLAVER
Dean for Research and Professional Development

Lynn Wolaver

Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

4 JUN 1982

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

ADA

INTERACTIVE DEBUGGING
COMPUTER DEBUGGING

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis involved the design and implementation of a skeletal interactive Ada debugger on the DEC-10 computer located at the Air Force Wright Avionics Laboratory. An analysis of current debugging technology was performed to formulat a basis for the debugger tool development.The tools implemented where breakpoints, single step and multi-step execution, display and modify program variables, as well as other miscellaneous options. Two conclusions were developed as the result of this thesis effort. First, because

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

of the lack of information on current software debugging methods, I have concluded that more emphasis is needed in techniques and tools for debugging of programs. Second, I have concluded that more emphasis is needed in the human interfacing techniques.